O'REILLY®

TURING 图灵程序设计丛书



函数式编程思维

Functional Thinking

[美] Neal Ford 著
郭晓刚 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍

郭晓刚

自命为平庸的开发者、失败的创业者和热情的翻译者。爱好多语言编程。在InfoQ中文站担任编辑近十年。有数本译作。全职照顾两岁的孩子，用20%的睡眠时间翻译技术书籍。

TURING 图灵程序设计丛书

函数式编程思维

Functional Thinking

[美] Neal Ford 著
郭晓刚 译

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（ＣＩＰ）数据

函数式编程思维 / (美) 福特 (Ford,N.) 著 ; 郭晓
刚 译. -- 北京 : 人民邮电出版社, 2015.8
（图灵程序设计丛书）
ISBN 978-7-115-40041-3

Ⅰ. ①函… Ⅱ. ①福… ②郭… Ⅲ. ①函数－程序设
计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2015)第176648号

内 容 提 要

本书脱离特定的语言特性，关注各种 OOP 语言的共同实践做法，展示如何通过函数式编程解决问题。知名软件架构师 Neal Ford 展示了不同的编程范式，帮助我们完成从 Java 命令式编程人员，到使用 Java、Clojure、Scala 的函数式编程人员的完美转变，建立对函数式语言的语法和语义的良好理解。

本书适合 Java、Clojure、Scala 及其他想要提高工作效率、关注函数式编程的程序员阅读。

◆ 著　　[美] Neal Ford
译　　郭晓刚
责任编辑　岳新欣
执行编辑　冯雪松
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：10.25
字数：242千字　　　　2015年 8 月第 1 版
印数：1 – 3 500册　　　　2015年 8 月北京第 1次印刷

著作权合同登记号　图字：01-2015-2577号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

译者序

函数式编程不是屠龙技。过去在一般开发者的认识里，函数式编程是一种仅仅存在于某些偏门语言里的学究气的概念。然而我们观察当今的主流语言，会发现函数式编程已经成为了标配，唯其存在形式发生了变化，从固执于“纯”函数式语言，转变为让一些关键的函数式特征或深或浅地融入到各式语言中去。

函数式编程的普及趋势，我以为主要应该归因于纯函数、一等函数、高阶函数等特征迎合了人们提高语法表现力和解决大规模并发问题的需要。函数式编程进入主流语言，意味着我们实际上已经在不同程度地使用着函数式编程。比如，你不一定用 F#，但 LINQ 实在是太方便了；你可能觉得 Clojure 太怪异，但 map、filter、reduce 任何时候都是必备的利器。

不同语言的函数式能力可以有很大的差别。那么在一些只能迂回模拟个别函数式特征的语言里面，去谈论函数式编程是否有意义？我对同行提到这本书用 Java 8 来解说函数式编程的时候，立即被编出了“只有这样才能写一本书”的笑话。笑点显然是因为用 Haskell、Lisp 来解说的话，写一章就够了。作者 Neal Ford 大概有不一样的看法，因为他故意用了 Scala、Clojure、Groovy、Java 8 这些函数式程度各异的语言，乃至在 Java 5 的极端环境下的 Functional Java 框架来证明，即使只是函数式编程的一个很小的子集，已经能够满足很大一部分需要，发挥很大的作用。毕竟，不管语法和实现上如何笨拙，函数式编程为我们开启的是另一个广阔的思考维度。不负责任地说，就算只学到了 map、filter、reduce 三板斧，你花在这本书上的时间都是值得的。

那么，要不要来学一学函数式编程呢？我想，开发者总不能比 Java 进步得还慢吧。

我把这本书翻译完了，而且，我敢保证，书里面没有一句话是你看不懂需要去翻原文的。把一本书从头到尾好好地译完，这件事情就算做过再多次，仍然值得我大大地夸一下自己，特别是我同时还要照顾两岁的郭寄傲小朋友。我的孩子要尝试 10 次、20 次才肯接受一种新的食物。我们接受一种新的范式，大概不会比这个简单。

绕了一个大圈子，我其实想说：靡不有初，鲜克有终。请不要只是买了这本书，而是真的学会函数式思维吧!

郭晓刚
2015 年 7 月

前言

我第一次认真研究函数式编程是在 2004 年。当时我受到 .NET 平台上一些非常规语言的吸引，开始摆弄 Haskell 和若干早于 F# 的 ML 家族语言。到了 2005 年，我开始在一些会议上做关于“.NET 和函数式语言”的演讲，不过那时候的语言多半还只是概念性的，即使说成是“玩具”也不为过。但不管怎么说，能够试探在一种新的编程思维范式下推演铺陈的可能性，已然令我乐在其中，而且这段经历改变了我在常规语言里对一些问题的处理方法。

2010 年我再次涉足这个研究领域，是因为目睹当时崛起的一批语言，例如 Java 生态圈里的 Clojure 和 Scala，一下子让我重温了五年前亲历的那些函数式世界的精妙所在。于是我在一个午后打开维基百科，顺着链接一页一页地翻阅着，半天时间下来，我已经完全沉迷其中。就这样，我一头钻入函数式编程的世界，开始了走遍各种思维分枝的探索历程。作为研究的成果，我于 2011 年在波兰举办的“33rd Degree Conference”大会（http://33degree.org/）上第一次做了题为“函数式编程思维”的演讲，又在 IBM developerWorks 网站上开设了同名的系列文章（http://dwz.cn/dev-works-ft-series）。在接下来的两年时间里，我按照每个月写一篇文章的进度，制订对函数式编程的研究和探索计划，并且坚持了下来。至今，我的函数式编程思维的演讲仍在继续，并且根据反馈不断完善。

这本书是对“函数式编程思维”演讲和系列文章中所有观点的总结。我发觉磨砺素材最好的办法是将之反复地呈现给观众，因为我每次做演讲或者写文章都会学到一些新东西。有些关联或者共性只有深入研究和被迫思考（截稿时间特别能让人集中精神！）之后，才会发现。

我在上一本书 *Presentation Patterns*（http://presentationpatterns.com/）中说过视觉象征对于会议演讲的重要性。因此我在做“函数式编程思维”演讲的时候，特意用了黑板和粉笔的形象（来引申出与函数式编程概念的数学联系）。到演讲结束的时候，我会呈现一张半截粉笔摆在黑板下方的图片，暗示观众自己拿起这半截粉笔，继续探索演讲中提到的观点。

我做的演讲，写的系列文章以及这本书，目的都是想针对那些在命令式的、面向对象的语言中浸淫已久的开发者，用一种他们能够理解的方式来介绍函数式编程的核心观点。希望我提炼的这些观点能引发你的兴致，并且拿起粉笔来继续你自己的探索。

——Neal Ford，2014 年 6 月于亚特兰大

本书结构

本书每一章都会演示函数式思维的例子。第 1 章“为什么”提供了概述和若干贯穿全书的思维转换的例子。第 2 章“转变思维”为程序员描绘了一个渐进的转变过程，让你从面向对象、命令式的观察角度过渡到函数式的观察角度。为了形象地展示这种思维转变，我分别用命令式风格和函数式风格来解决同一个常见问题以作对比。然后又通过一个详尽的案例分析来说明函数式的观察角度（以及若干辅助语法）如何帮助你向函数式的思维方式转变。

第 3 章“权责让渡”列举了一些可以放心托付给语言或运行时去处理的日常杂务。*状态*是 Michael Feathers 所谓的“不确定因素”之一，通常在非函数式的语言里需要直接明确地进行管理。闭包（closure）允许我们将一部分状态管理工作交托给运行时；我举了一些例子来说明这个状态处理机制背后的工作原理。这一章还会展示如何按照函数式思维在一些细节方面放权，例如把集合操作交给递归。这些思路将对代码重用的粒度产生影响。

第 4 章“用巧不用蛮”着重讨论两个延续“消灭不确定因素”精神的例子，它们利用运行时来缓存函数的结果，从而获得*缓求值*（laziness）的特性。很多函数式语言都包含“记忆”（memoization）特性（可能直接支持，也可能通过库，或者用一点小技巧就能实现），可以作为一种常用的性能优化手段。我在第 2 章“完全数分类器”例子的基础上比较了几种不同层次的优化手段，有手工进行的，也有利用语言提供的记忆特性来完成的。如果你想提前知道比赛的结果，记忆特性是最后的赢家。缓求值（lazy）的数据结构把运算推迟到最后时刻才去执行，这个特点让我们有机会换一个角度来看待各种数据结构。我演示了如何实现缓求值数据结构（甚至可以用非函数式语言来实现），以及如何利用语言已经具备的缓求值特性。

第 5 章“演化的语言”反映各种语言是怎样朝着加强函数式特征的方向演变的。本章还会讨论若干革命性的语言发展趋势，如操作符重载和方法调用之外的新的分派（dispatch）方式，讨论让语言去迎合问题（而不是反过来）的观点，以及 Option 等常见的函数式数据结构。

第 6 章“模式与重用”通过一些例子来展示解决问题的一般思路。我分析了传统的设计模式在函数式编程的世界里是怎样蜕变（或者消失）的。我还详细对比了通过继承和通过组合这两种代码重用方式，并从耦合的角度对它们进行了由表及里的分析。

第 7 章“现实应用”详细展示了 Java 开发工具包（JDK）新增的几项人们期待已久的函数式特性。从分析中可以看到，Java 8 也像别的语言一样接纳了函数式思维，它的高阶函数（即 lambda 块）用法就是表现之一。我还讨论了 Java 8 在保持向后兼容上使用的一些巧妙而优雅的手法。Stream API 是特别提到的一个发扬了函数式思维的亮点，它能够以描述性的语言简洁明了地表达工作流。最后我介绍了 Java 8 新增的 Option 结构，它解决了 null 返回值含义模糊的潜在问题。我还用了一些篇幅来讨论函数式架构和数据库的主题，分析函数式的视角怎样改变了它们的设计。

第 8 章“多语言与多范式”叙述了函数式编程对于当前这个多语言世界的影响。我们一直在各种项目中遭遇和容纳越来越多的语言。很多新的语言都是多范式（polyparadigm）的，同时支持若干种不同的编程模型。例如 Scala 支持面向对象编程和函数式编程。作为最后一章，我们探讨了活在一个有更多范式可以选择的世界里有什么好处和坏处。

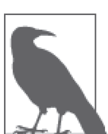
排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语或突出强调的内容。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。



该图标表示一般注记。

使用代码示例

补充材料（示例代码、练习等）可以从 https://github.com/oreillymedia/functional_thinking 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产

品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Functional Thinking* by Neal Ford (O'Reilly). Copyright 2014 Neal Ford, 978-1-449-36551-6."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：http://dwz.cn/functional-thinking。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

感谢 ThoughtWorks 大家庭，这是我能找到的最好的工作环境。感谢和我一起参加各种会议的讲师们，尤其是“No Fluff, Just Stuff”会议的讲师们，给了我许多思想的碰撞。感谢这些年来出席“函数式编程思维”演讲的观众，你们的反馈帮我磨砺了本书的素材。特别感谢本书的技术审阅人，他们给出了中肯的实质性建议。尤其感谢那些花时间提交勘误的早期读者，你们为我揭示了许多视而不见的晦涩之处。感谢数不清的朋友和家人充当了我坚实的后盾，特别感谢 John Drescher 在我们离家的时候帮忙照顾猫咪们。当然，还要感谢一直忍耐包容我的夫人 Candy，她早就不指望我能放下对编程的迷恋了。

第1章
为什么

我们用几分钟来想象一下自己是一名伐木工人，手里有林场里最好的斧子，因此你是工作效率最高的。突然有一天场里来了个推销的，他把一种新的砍树工具——链锯——给夸到了天上去。这人很有说服力，所以你也买了一把，不过你不懂得怎么用。你估摸着按照自己原来擅长的砍树方法，把链锯大力地挥向树干——不知道要先发动它。“链锯不过是时髦的样子货罢了”，没砍几下你就得出了这样的结论，于是把它丢到一边重新捡起用惯了的斧子。就在这个时候，有人在你面前把链锯给发动了……

学习一种全新的编程范式，困难并不在于掌握新的语言。毕竟能拿起这本书的读者，学过的编程语言少说也有一箩筐——语法不过是些小细节罢了。真正考验人的，是怎么学会用另一种方式去思考。

本书探讨函数式编程的话题，但重点并不放在函数式编程语言上。请别误会，我并不打算空谈理论，书里会有用很多种语言写成的大量代码，实际上整本书都是围绕着代码来展开的。用“函数式”的方式编写代码牵涉到诸多方面，我会用具体的例子来解说各方面的要旨，包括设计上的种种取舍、不同重用单元的作用等。比起语法，我更看重思路，因此解说会从 Java 语言入手，毕竟这是最大的开发者群体的最基本的共同语言，而且会掺杂 Java 8 和旧版 Java 的例子。我会尽可能地用 Java 语言（或其近亲）来解释函数式编程概念，仅仅用其他语言来演示一些独有的特性。

也许你对 Scala 和 Clojure 一点都不感兴趣，下半辈子能有现在用着的语言就心满意足了，可是你的语言并不会停下来，反而时刻都在变得更加函数式，也径直带着你一起。所以说，现在快来学学函数式编程范式吧，这样，当有一天（不是假如）函数式降临你日常使

用的语言的时候，你才知道如何驾驭它。我们不妨先了解一下，为什么所有的语言都日渐向函数式靠拢。

1.1 范式转变

计算机科学的进步经常是间歇式的，好思路有时搁置数十年后才突然间变成主流。举个例子，第一种面向对象的语言 Simula 67 是 1967 年发明的，可是直到 1983 年诞生的 C++ 终于流行起来以后，面向对象才真正成为主流。很多时候，再优秀的想法也得等待技术基础慢慢成熟。早年 Java 总被认为太慢，内存耗费太高，不适合高性能的应用，如今硬件市场的变迁把它变成了极具吸引力的选择。

函数式编程的发展轨迹与面向对象编程十分相似，它也是诞生在学院里，然后用几十年的时间悄悄浸染了所有的现代编程语言。不过，仅仅在语言里加入一些新语法，并不足以让开发者完全发挥出这种新思维的全部力量。

我们的讨论可以从两种风格的对比开始，尝试分别用传统编程风格（命令式的循环）和函数式特征更明显的方式来解决同一道题目。这道题目出自计算机科学史上的著名事件，是当年 *Communications of the ACM* 杂志“Programming Pearls”专栏的作者 Jon Bentley 向计算机科学先驱 Donald Knuth 提出的挑战。涉猎过文本操作的开发者会很熟悉这道题目：*读入一个文本文件，确定所有单词的使用频率并从高到低排序，打印出所有单词及其频率的排序列表。*对于问题中的词频统计部分，我给出了一个“传统”Java 的解答，见例 1-1。

例 1-1 词频统计的 Java 实现

```
public class Words {
    private Set<String> NON_WORDS = new HashSet<String>() {{
        add("the"); add("and"); add("of"); add("to"); add("a");
        add("i"); add("it"); add("in"); add("or"); add("is");
        add("d"); add("s"); add("as"); add("so"); add("but");
        add("be"); }};

    public Map wordFreq(String words) {
        TreeMap<String, Integer> wordMap = new TreeMap<String, Integer>();
        Matcher m = Pattern.compile("\\w+").matcher(words);
        while (m.find()) {
            String word = m.group().toLowerCase();
            if (! NON_WORDS.contains(word)) {
                if (wordMap.get(word) == null) {
                    wordMap.put(word, 1);
                }
                else {
                    wordMap.put(word, wordMap.get(word) + 1);
                }
            }
        }
        return wordMap;
```

```
    }
}
```

例 1-1 首先建立了一个“虚词”（nonwords）的集合（包括冠词和其他起连接作用的词），然后实现了 `wordFreq()` 方法。方法中首先建立一个 `Map` 结构来容纳由单词和词频组成的键值对，接着构造了一个用来识别单词的正则表达式。接下来的大段篇幅逐一遍历所有找到的单词，将首次遇到的单词添入 `Map` 结构，将重复遇到的单词的出现次数加 1。对于提倡以步进方式处理集合（如例中正则表达式的匹配结果）遍历的语言来说，这是司空见惯的编码风格。

Java 8 新增了 `Stream` API 和以 lambda 块方式实现的高阶函数（后文将会详细介绍），我们利用这些新的编程手段来改写上面的例子，就得到例 1-2。

例 1-2　词频统计的 Java 8 实现

```
private List<String> regexToList(String words, String regex) {
    List wordList = new ArrayList<>();
    Matcher m = Pattern.compile(regex).matcher(words);
    while (m.find())
    wordList.add(m.group());
    return wordList;
}

public Map wordFreq(String words) {
    TreeMap<String, Integer> wordMap = new TreeMap<>();
    regexToList(words, "\\w+").stream()
            .map(w -> w.toLowerCase())
            .filter(w -> !NON_WORDS.contains(w))
            .forEach(w -> wordMap.put(w, wordMap.getOrDefault(w, 0) + 1));
    return wordMap;
}
```

在例 1-2 里，我将正则表达式的匹配结果转换为 stream，更方便后续执行互相独立的几项操作：将所有的单词条目转换为小写，滤除虚词，计算余下单词的词频。我把 `regexToList()` 方法经由 `find()` 产生的匹配结果集合转换成 stream，这是为了让后续的操作能够像我们考虑问题的方式一样，做完一步再去做下一步。虽然将命令式风格的例 1-1 改为对集合进行三次循环遍历（第一遍把所有的单词变成小写，第二遍滤除虚词，第三遍计算词频）也能达成目的，但这种写法的效率会惨不忍睹。例 1-1 在一个迭代块里完成三项操作，这是牺牲了代码的清晰来换取执行性能。哪怕这种牺牲再稀松平常，我总是不情愿的。

Clojure 语言（http://clojure.org/）的发明人 Rich Hickey 在 Strange Loop 会议上做过一堂题为“Simple Made Easy” 的 演 讲（http://www.infoq.com/presentations/Simple-Made-Easy），他翻出了一个已经很少用到的老词——“交织”（complect）：穿插缠绕地合为一体，使错综复杂。命令式编程风格常常迫使我们出于性能考虑，把不同的任务交织起来，以便能够用一次循环来完成多个任务。而函数式编程用 `map()`、`filter()` 这些高阶函数把我们解放

出来，让我们站在更高的抽象层次上去考虑问题，把问题看得更清楚。后文我们将看到许多函数式思维破解交织现象的例子。

1.2　跟上语言发展的潮流

如果我们关注各种语言的发展情况就会发现，所有的主流语言都在进行函数式方面的扩充。早走一步的 Groovy 已经具备了丰富的函数式特性，包括像“记忆”（memoization，指运行时自动缓存函数返回值的能力）这样的高级特性在内。随着 lambda 块（也就是高阶函数）被纳入 Java 8，Java 语言也终于披挂上函数式的武器。JavaScript，这种也许算得上使用最为广泛的语言，本身就拥有不少函数式特性。就连最老成持重的 C++ 语言，也在 2011 年版的语言标准里增加了 lambda 块，引人关注的 Boost.Phoenix（http://dwz.cn/phoenix-library）等类库，更是透露出函数式思潮已经对 C++ 语言有了更深入的影响。

不论你用的是 Clojure 这类新语言，还是日常相伴的老语言，都有可能遇到相关的特性，而只有学会这些新的编程范式，你才能从容地利用它们。我会在第 2 章讨论如何转变思维，运用这些先进的工具去大展拳脚。

1.3　把控制权让渡给语言/运行时

在计算机科学短短的发展历史上，有时候会从技术主流分出一些枝杈，有源于实务界的，也有源于学术界的。例如在 20 世纪 90 年代个人电脑大发展的时期，第四代编程语言（4GL）也出现了爆发式的流行，涌现了 dBASE、Clipper、FoxPro、Paradox 等不可胜数的新语言。这些语言的卖点之一是比 C、Pascal 等第三代语言（3GL）更高层次的抽象。换言之，4GL 下的一行命令，3GL 可能要用很多行才写得出来，因为 4GL 自带了更丰富的编程环境。像从磁盘读取流行的数据库格式这样的功能，4GL 天生就具备，并不需要使用者特意去实现。

函数式编程也是这样一根横生出来的枝杈，是学术界那些乐于为新思路和新范式寻找表达手段的计算机科学家们的发明。分出来的枝杈偶尔会重新汇入主流，函数式编程当前正好是这种情况。函数式语言不仅在 Java 虚拟机（JVM）平台上迅速地崭露头角，例如最有代表性的 Scala 和 Clojure 语言，.NET 平台也不例外，F# 已经是堂堂正正的平台一员。那么，为什么所有的平台都在拥抱函数式编程呢？

20 世纪 80 年代早期，我还在上大学的时候，用的编程环境叫作 Pecan Pascal。Pecan Pascal 的独门绝技是可以在 Apple][和 IBM PC 上运行相同的 Pascal 代码。为了做到这一点，Pecan 的工程师祭出了神秘的“字节码”（bytecode）。在编译的时候，开发者写下的 Pascal 源代码会被编译成这种在“虚拟机”上执行的“字节码”，而“虚拟机”在每一种运行平台上都有专门的原生实现。Pecan Pascal 用起来让人痛不欲生。就算最简单的编程习题，

编译出来的代码都慢得无法忍受。当时的硬件水平还没有准备好迎接这样的挑战。

Pecan Pascal 被淘汰了，但它的架构我们都很熟悉。十年之后 Sun 发布了采用同样设计的 Java，在 20 世纪 90 年代中期的硬件环境下勉力取得了成功。Java 还带来了其他一些救开发者于水火的特性，自动垃圾收集即是其中之一。我从此再也不想碰那些没有垃圾收集的语言。亲身经历告诉我，最好还是把时间花在更高层次的抽象上，多考虑怎样解决复杂的业务场景，少去费心复杂的底层运作。我为 Java 纾解了人工管理内存的痛苦而欣喜，同时期冀在别的方面也能找到这样的利器。

人生苦短，远离 `malloc`。

随着时间的推移，开发者们越来越多地把乏味单调的任务托付给语言和运行时。对于我日常编写的应用程序类型来说，失去对内存的直接控制没什么可惋惜的，放弃这些反而让我能够专注于更重要的问题。Java 接管内存分配减轻了我们的负担，函数式编程语言让我们用高阶抽象从容取代基本的控制结构，也有着同样的意义。

将琐碎的细节交托给运行时，令繁冗的实现化作轻巧，这样的例子本书中比比皆是。

1.4 简洁

Working with Legacy Code 的作者 Michael Feathers 用寥寥数语（https://twitter.com/mfeathers/status/29581296216）捕捉到了函数式抽象和面向对象抽象的关键区别：

> 面向对象编程通过封装不确定因素来使代码能被人理解；函数式编程通过尽量减少不确定因素来使代码能被人理解。
>
> ——Michael Feathers

请回想一下你熟悉的封装、作用域、可见性等面向对象编程（OOP）构造，这些机制的存在意义，都是为了精细地控制谁能够感知状态和改变状态。而当涉及多线程的时候，对状态的控制就更复杂了。这些机制就属于 Michael Feathers 所谓的“不确定因素”（moving parts）。大多数函数式语言在这个问题上采取了另一种做法，它们认为，与其建立种种机制来控制可变的状态，不如尽可能消灭可变的状态这个不确定因素。其立论的根据是这样的：假如语言不对外暴露那么多有出错可能的特性，那么开发者就不那么容易犯错。我会展示各种例子来说明函数式编程是怎样消除变量、抽象和其他不确定因素的。

在面向对象的命令式编程语言里面，重用的单元是类和类之间沟通用的消息，这些都可以用类图（class diagram）来表述。这个领域的代表性著作《设计模式：可复用面向对象

软件的基础》（*Design Patterns: Elements of Reusable Object-Oriented Software*，作者 Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides）就在每一个模式的说明里都附上了至少一幅类图。OOP 的世界提倡开发者针对具体问题建立专门的数据结构，相关的专门操作以"方法"的形式附加在数据结构上。函数式编程语言实现重用的思路很不一样。函数式语言提倡在有限的几种关键数据结构（如 `list`、`set`、`map`）上运用针对这些数据结构高度优化过的操作，以此构成基本的运转机构。开发者再根据具体用途，插入自己的数据结构和高阶函数去调整机构的运转方式。

我们来分析下面截取自例 1-2 的片段：

```
regexToList(words, "\\b\\w+\\b").stream()
                .filter(w -> !NON_WORDS.contains(w))
```

这里为了取得列表的一个子集而调用了 `filter()` 方法，并向 `filter()` 方法传入已被转换为 stream 的列表内容，以及定义了筛选条件的高阶函数（即行中裹上了语法糖衣的 `(w → !NON_WORDS.contains(w))`）。运转机构高效率地按照指定的条件实行筛选，返回筛选后的列表。

比起一味创建新的类结构体系，把封装的单元降低到函数级别，更有利于达到细粒度的、基础层面的重用。反面例子如 Java 世界的数十种 XML 类库，每一种都有自己定义的内部数据结构。相比之下，Clojure 就享受到了使用高层次抽象的好处。不久前 Clojure 库中的 `map` 方法经过创造性的重写，获得了自动并行的能力，也就是说，所有 Clojure 开发者不需要动一行代码，就自动享受到了 map 操作的性能提升。

函数式程序员喜欢用少数几个核心数据结构，围绕它们去建立一套充分优化的运转机构。面向对象程序员喜欢不断地创建新的数据结构和附属的操作，因为压倒一切的面向对象编程范式就是建立新的类和类间的消息。把所有的数据结构都封装成类，一方面压制了方法层面的重用，另一方面鼓励了大粒度的框架式的重用。函数式编程的程序构造更方便我们在比较细小的层面上重用代码。

例 1-3 取自为 Java 提供大量辅助工具类的 Apache Commons（http://commons.apache.org/proper/commons-lang）框架，请观察下面的 `indexOfAny()` 方法。

例 1-3　取自 Apache Commons 工具类 `StringUtils` 的 `indexOfAny()` 方法

```
// 来源于Apache Commons Lang,http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars) {
    if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) { ❶
        return INDEX_NOT_FOUND;
    }
    int csLen = str.length();                    ❷
    int csLast = csLen - 1;
    int searchLen = searchChars.length;
    int searchLast = searchLen - 1;
```

```
        for (int i = 0; i < csLen; i++) {           ❸
            char ch = str.charAt(i);
            for (int j = 0; j < searchLen; j++) {    ❹
                if (searchChars[j] == ch) {          ❺
                    if (i < csLast && j < searchLast && CharUtils.isHighSurrogate(ch)) {
                        if (searchChars[j + 1] == str.charAt(i + 1)) {
                            return i;
                        }
                    } else {
                        return i;
                    }
                }
            }
        }
        return INDEX_NOT_FOUND;
    }
```

❶ 防范参数错误。

❷ 初始化。

❸ 外层迭代。

❹ 内层迭代。

❺ 判断多组条件。

`indexOfAny()` 方法的参数是一个 `String` 和一个数组，它会在 `String` 中查找数组里的字符，并返回任意一个字符第一次出现的索引位置。其文档中举了一些例子来说明输入与输出的关系，见例 1-4。

例 1-4　`indexOfAny()` 的用法示例

```
StringUtils.indexOfAny("zzabyycdxx",['z','a']) == 0
StringUtils.indexOfAny("zzabyycdxx",['b','y']) == 3
StringUtils.indexOfAny("aba", ['z'])           == -1
```

我们看到，字符串 `zzabyycdxx` 中第一次出现字符 `z` 或 `a` 是在索引位置 `0`，第一次出现字符 `b` 或 `y` 是在索引位置 `3`。

这个问题的实质可以表述为：对于 `searchChars` 中的任意字符，在目标字符串中查找该字符第一处匹配的索引位置。假如换成 Scala 语言，同样的方法实现起来要直接得多，请看例 1-5 的 `firstIndexOfAny` 方法。

例 1-5　Scala 实现的 `firstIndexOfAny()`

```
def firstIndexOfAny(input : String, searchChars : Seq[Char]) : Option[Int] = {
  def indexedInput = (0 until input.length).zip(input)
  val result = for (pair <- indexedInput;
                    char <- searchChars;
                    if (char == pair._2)) yield (pair._1)
  if (result.isEmpty)
    None
```

```
    else
      Some(result.head)
  }
```

在本例中，我为输入字符串制作了一个添加了索引的版本。Scala 的 `zip()` 方法将（从 0 到输入字符串长度值的）数字集合与 `String` 对象中所含字符的集合对位结合，组成一个新的、由数字和字符对构成的集合。例如当输入字符串为 `zabycdxx` 时，`indexedInput` 将取值为 `Vector ((0,z), (1,a), (2,b), (3,y), (4,c), (5,d), (6,x), (7,x))`。`zip` 方法得名于它像拉链（zipper）一样让两个集合对齐咬合在一起。

准备好索引集合之后，我使用 Scala 的 for comprehension 首先查看待搜索字符的集合，然后取出索引集合中的索引字符对。由于 Scala 允许快捷访问集合的元素，所以我可以直接将当前搜索的字符与集合的第二个元素进行比较（`(if (char == pair._2))`）。如果两个字符相同，那么返回索引字符对的索引部分（`pair._1`）。

`null` 的存在是 Java 语言的一大混乱来源：它到底是一个有效的返回值，还是表明返回值缺失了？包括 Scala 在内的很多函数式语言通过 `Option` 类来避免这种语义上的含混，其取值要么是表示没有返回值的 `None`，要么是容纳了返回值的 `Some`。因为例 1-5 的需求只要求找到第一处匹配，所以我返回了结果集合的第一个元素 `result.head`。

从原本需求的第一处匹配改为返回所有的匹配是轻而易举的事情。只要修改一下返回类型，并去掉返回值外面的包装就可以了，修改后的代码见例 1-6。

例 1-6　返回匹配项的一个缓求值列表

```
def indexOfAny(input : String, searchChars : Seq[Char]) : Seq[Int] = {
  def indexedInput = (0 until input.length).zip(input)
  for (pair <- indexedInput;
       char <- searchChars;
       if (char == pair._2)) yield (pair._1)
}
```

修改后的 API 去掉了限制，让用户自己决定需要多少个返回值。执行 `firstIndexOfAny("zzabyycdxx", "by")` 会得到返回值 `3`，而 `indexOfAny("zzabyycdxx", "by")` 的返回值则是 `Vector(3, 4, 5)`。

第 2 章

转变思维

学习一门新的编程语言一点都不难，你只要知道怎么把熟悉的概念用新的语法表达出来就行了。比如说你打算学 JavaScript，那么第一步会去找份资料，看看 JavaScript 是怎么表达 `if` 语句的。通常程序员可以通过套用自己已经在别的语言中掌握的知识来学习新的语言。与之相比，学习一种新的范式是困难的——我们必须学会为熟悉的问题找到新的解答方法。

换用 Scala、Clojure 之类的函数式编程语言并不是写出函数式代码的必要条件，转变我们看待问题的角度才是必不可少的。

2.1　普通的例子

当垃圾收集成为主流，一下子将若干难以调试的错误类别连根拔起，程序员也因为运行时接管了复杂且容易出错的内存管理而获得解脱。函数式编程希望在算法编写上给予程序员同样的帮助，一方面程序员得以在更高的抽象层次上工作，另一方面运行时也有了执行复杂优化的自由空间。开发者从中获得的好处体现在更低的复杂性和更高的性能，这点与垃圾收集相同，不过，函数式编程对个人的影响更直接，因为它改变的是你的解答思路。

2.1.1　命令式解法

命令式编程是按照“程序是一系列改变状态的命令”来建模的一种编程风格。传统的 `for` 循环是命令式风格的绝好例子：先确立初始状态，然后每次迭代都执行循环体中的一系列命令。

为了形象说明命令式编程与函数式编程的差异，我会从一个普通的问题和它的命令式解法说起。假设我们有一个名字列表，其中一些条目由单个字符构成。现在的任务是，将除去单字符条目之外的列表内容，放在一个逗号分隔的字符串里返回，且每个名字的首字母都要大写。实现这个算法的 Java 代码见例 2-1。

例 2-1　典型的公司业务处理例子（Java 实现）

```
package com.nealford.functionalthinking.trans;

import java.util.List;

public class TheCompanyProcess {
    public String cleanNames(List<String> listOfNames) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < listOfNames.size(); i++) {
            if (listOfNames.get(i).length() > 1) {
                result.append(capitalizeString(listOfNames.get(i))).append(",");
            }
        }
        return result.substring(0, result.length() - 1).toString();
    }

    public String capitalizeString(String s) {
        return s.substring(0, 1).toUpperCase() + s.substring(1, s.length());
    }
}
```

由于我们处理例 2-1 的问题时必定要遍历整个列表，那么最方便下手操作的地方，自然就是在一个命令式循环的内部。每迭代一个名字，我们都检查它的长度是否大于一个字符的保留门槛，然后调整其首字母为大写后，连同作为分隔符的逗号一起，追加到 `result`。最后一个名字不应该有尾随的逗号，所以我们从最后的返回值里去掉了这个多余的分隔符。

命令式编程鼓励程序员将操作安排在循环内部去执行。本例中我做了三件事：filter，筛选列表，去除单字符条目；transform，变换列表，使名字的首字母变成大写；接着是 convert，转换列表，得到单个字符串。这三种操作可以说是我们在列表上施展的“三板斧”。在命令式语言里，这三种操作都必须依赖于相同的低层次机制（对列表进行迭代）。而函数式语言为这些操作提供了针对性的辅助手段。

2.1.2　函数式解法

函数式编程将程序描述为表达式和变换，以数学方程的形式建立模型，并且尽量避免可变的状态。函数式编程语言对问题的归类不同于命令式语言。如上一小节所列的几种操作（filter、transform、convert），每一种都作为一个逻辑分类由不同的函数所代表，这些函数实现了低层次的变换，但依赖于开发者定义的高阶函数作为参数来调整其低层次运转机构的运作。于是，上一小节的问题可以概念性地表达为例 2-2 的伪代码。

例 2-2　伪代码表示的“公司业务处理过程”

```
listOfEmps
    -> filter(x.length > 1)
    -> transform(x.capitalize)
    -> convert(x + "," + y)
```

函数式语言可以帮助我们轻松搭建出上面的概念性解答模型，同时又不必操心各种实现细节。

假如我们用 Scala 来实现例 2-1 的公司业务处理过程，将会是例 2-3 的样子。

例 2-3　函数式的处理过程（Scala 实现）

```
val employees = List("neal", "s", "stu", "j", "rich", "bob", "aiden", "j", "ethan",
         "liam", "mason", "noah", "lucas", "jacob", "jayden", "jack")

val result = employees
  .filter(_.length() > 1)
  .map(_.capitalize)
  .reduce(_ + "," + _)
```

例 2-3 的 Scala 代码除了补充一些必要的实现细节，其写法简直和例 2-2 的伪代码如出一辙。拿到名字列表，首先进行筛选，消去单字符条目。筛选操作的输出结果紧接着被送入 `map` 函数，让 `map` 函数对输入集合的每个元素执行参数内提供的代码块，并返回变换后的集合。最后，`map` 的输出集合被送入 `reduce()` 函数，由 `reduce()` 函数根据参数内作为规则传入的代码块，将集合元素逐一拼合起来。例中我们传入了用逗号来连接前两个元素的规则。在调用例中三个函数的时候，参数取什么名字无关紧要，正好 Scala 允许我们跳过命名步骤，直接以下划线来代替。其中在调用 `reduce()` 函数的时候，我们其实按照方法的签名传入了两个参数，虽然两个位置上都写着相同的参数占位符——下划线。

挑选 Scala 作为演示的第一种语言，除了语法上相似，还因为 Scala 对于我们要演示的几个概念都采用了与业界一致的命名。实际上 Java 8 也具备作出函数式解答所需要的语言特性，且其实现各方面都与 Scala 版本十分近似，请看例 2-4。

例 2-4　Java 8 实现的处理过程

```
public String cleanNames(List<String> names) {
    if (names == null) return "";
    return names
            .stream()
            .filter(name -> name.length() > 1)
            .map(name -> capitalize(name))
            .collect(Collectors.joining(","));
}

private String capitalize(String e) {
    return e.substring(0, 1).toUpperCase() + e.substring(1, e.length());
}
```

例 2-4 用 collect() 方法取代了 reduce()，原因是它操作 Java 的 String 类的效率更高；collect() 是 Java 8 针对某些情形而提供的 reduce() 的特殊实现。除了这一点点差别，上面的代码与例 2-3 的 Scala 实现极其相似。

如果我们担心某些列表元素可能为 null，那么只要在 stream 后面多加一条检查就可以了：

```
return names
        .stream()
        .filter(name -> name != null)
        .filter(name -> name.length() > 1)
        .map(name -> capitalize(name))
        .collect(Collectors.joining(","));
```

Java 运行时会聪明地将 null 检查和针对长度的筛选合并成一次操作，这样既不妨碍我们把意图表达清楚，又不损失代码的执行效率。

Groovy 语言也具备实现例 2-2 模型所需的特性，不过命名上更接近 Ruby 等脚本语言。Groovy 版的实现代码请看例 2-5。

例 2-5　Groovy 实现的处理过程

```
public static String cleanUpNames(listOfNames) {
  listOfNames
      .findAll { it.length() > 1 }
      .collect { it.capitalize() }
      .join ','
}
```

例 2-5 的代码结构上与例 2-3 的 Scala 实现基本一致，只是方法名称和参数占位符不一样。Groovy 的 findAll 方法对集合中的元素执行参数里传入的代码块，只留下结果为 true 的元素。Groovy 也像 Scala 一样允许开发者简写只带一个参数的代码块，它规定用 it 关键字来代表这个唯一的参数，无需定义。Groovy 的 collect 方法相当于前面的 map，负责对集合中的每个元素执行参数里传入的代码块。join() 函数的功能是用参数中指定的分隔符，把一个字符串集合串接起来，拼成单一的字符串，正好符合我们的需要。

Clojure 是一种函数式语言，它的函数命名上自然更传统一些。请看例 2-6。

例 2-6　Clojure 实现的处理过程

```
(defn process [list-of-emps]
  (reduce str (interpose ","
    (map s/capitalize (filter #(< 1 (count %)) list-of-emps)))))
```

不熟悉 Clojure 语法的话，例 2-6 的代码结构可能不太好分辨。Lisp 家族的 Clojure 是“由内向外”执行的，因此起点其实在最后一个参数值 list-of-emps。Clojure 的 (filter a b) 函数接受两个参数：作为筛选条件的函数（例中为匿名函数）和将要被筛选的集合。假如我们愿意，第一个参数也可以写成完整的函数定义 (fn [x] (< 1 (count x)))，不过

Clojure 允许我们使用更简短的匿名函数形式 `#(< 1 (count %))`。这一步筛选操作的结果也像前面的例子一样，是一个消除了部分元素的小一点的集合。

`(map a b)` 函数的第一个参数是变换函数，第二个参数是待变换的集合，也就是上一步 `(filter )` 操作的返回值。我们可以专门定制一个函数来作为 `(map )` 的第一个参数，不过既然任何单参数的函数都符合 `(map )` 的要求，我们直接用能够满足需求的 Clojure 内建函数 `capitalize` 即可。最后，`(map )` 操作的输出成为下一步 `(reduce )` 操作的集合参数。`(reduce )` 的第一个参数是负责拼合字符串的 `(str )` 函数，`(str )` 作用于 `(interpose )` 函数的返回值，而 `(interpose )` 负责在 `(map )` 返回集合的元素之间插入它的第一个参数指定的分隔符。

面对这样嵌套了一层又一层的函数结构，就连经验丰富的开发者也会痛苦不堪。幸好 Clojure 有一些宏可以帮助我们把这样的结构"捋顺"，变成更方便阅读的顺序。请看例 2-7，它与例 2-6 功能上完全一致。

例 2-7　通过 `thread-last` 宏改善代码的可读性

```
(defn process2 [list-of-emps]
  (->> list-of-emps
       (filter #(< 1 (count %)))
       (map s/capitalize)
       (interpose ",")
       (reduce str)))
```

Clojure 的 thread-last 宏（即 `->>` 符号）针对的是非常常见的各种集合变换操作，它把典型的 Lisp 书写顺序颠倒了过来，重整为更自然的从左到右的阅读顺序。例 2-7 中我们首先看到的是集合本身（`list-of-emps`），然后才是依次作用于前一个语法单元（form）的连串变换操作。Lisp 灵活的语法正是它最强大的武器之一：什么时候可读性变差了，我们就调整语法去满足可读性。

上面提到的语言都已经具备了函数式编程的关键概念。向函数式思维靠拢，意味着我们逐渐学会何时何地应该求助于这些更高层次的抽象，不要再一头扎到实现细节里去。

学会用更高层次的抽象来思考有什么好处？首先，会促使我们换一种角度去归类问题，看到问题的共性。其次，让运行时有更大的余地去做智能的优化。有时候，在不改变最终输出的前提下，调整一下作业的先后次序会更有效率（例如减少了需要处理的条目）。第三，让埋头于实现细节的开发者看到原本视野之外的一些解决方案。举个例子，例 2-1 的 Java 实现要改成多线程的话，需要的工作量可不小。由于我们自己控制着低层次的迭代细节，那么线程相关的代码也就只好由我们自己动手穿插进去。可是换作 Scala 的实现，我们只要在 stream 上多调用一次 `par` 方法就可以了，请看例 2-8。

例 2-8　Scala 实现的并行化处理过程

```
val parallelResult = employees
```

```
    .par
    .filter(_.length() > 1)
    .map(_.capitalize)
    .reduce(_ + "," + _)
```

Java 8 实现要达到相同的并行化效果，也只需要做几乎一样的简单改动，如例 2-9 所示。

例 2-9　Java 8 实现的并行化处理过程

```
public String cleanNamesP(List<String> names) {
    if (names == null) return "";
    return names
            .parallelStream()
            .filter(n -> n.length() > 1)
            .map(e -> capitalize(e))
            .collect(Collectors.joining(","));
}
```

Clojure 同样只需简单替换，就能够将一般的集合变换操作不动声色地并行化。我们在更高的抽象层次上做事情，运行时才好去优化低层次的细节。编写带垃圾收集的工业级虚拟机实在是一项异常复杂的任务，开发者乐得交出这方面的职责。另一边的 JVM 工程师则尽力封装起垃圾收集，让它从开发者的日常考虑事项中消失，大大减轻了开发者的负担。

`map`、`reduce`、`filter` 等函数式操作也存在类似的互利关系。Clojure 下的 Reducers 扩展库（http://dwz.cn/reducers-library）就是一个绝佳的例子。其作者 Rich Hickey 以库的形式对 Clojure 语言进行了扩展，提供了新版本的 `vector` 和 `map` 实现（以及用来转换原版 `vector` 和 `map` 的新的 `fold` 函数），他的实现在内部运用 Java 的 Fork/Join 框架来完成对集合的并行处理。Clojure 的一个重要的卖点，就是它从一般开发者可见的层面抹去了并发的麻烦，就好像 Java 消除了垃圾收集的麻烦一样。而使用 Clojure 的开发者自觉地用 `map` 来取代原始的迭代，因而自动享受到新版本的能力提升。

多从结果着眼，少纠结具体的步骤。

不要再让那些迭代、变换、化约如何进行的低层次细节占据你的思维，多想想哪些问题其实可以归结为这几样基本操作的排列组合吧。

我们还可以举一个例子来说明怎样从一个命令式的解法过渡到函数式的答案，这一次我们用完美数（perfect number）的分类问题来做说明。

2.2　案例研究：完美数的分类问题

古希腊数学家 Nicomachus 发明了一种自然数的分类方法，任意一个自然数都唯一地被归类为*过剩数*（abundant）、*完美数*（perfect）或*不足数*（deficient）。一个完美数的真约数（即除了自身以外的所有正约数）之和，恰好等于它本身。例如 6 是一个完美数，因为它的约数是 1、2、3，而 6 = 1 + 2 + 3；28 也是一个完美数，因为 28 = 1 + 2 + 4 + 7 + 14。根据完美数的定义，我们可以得到如表 2-1 所示的分类规则。

表2-1：自然数分类规则

完美数	真约数之和 = 数本身
过剩数	真约数之和 > 数本身
不足数	真约数之和 < 数本身

实现中用到一个数学概念，*真约数和*（aliquot sum），其定义就是除了数本身之外（一个数总是它本身的约数），其余正约数的和。之所以不用“正约数和”来表述，是为了稍稍简化判定完美数时的比较语句：`aliquotSum == number` 要比 `sum - number == number` 易读一些。

2.2.1　完美数分类的命令式解法

我们的分类程序在使用中很可能需要对同一个数字进行多次分类，因此实现的时候有必要考虑这种情况。带着这样的需求，我们得出如例 2-10 所示的 Java 实现。

例 2-10　完美数分类的 Java 实现

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

public class ImpNumberClassifierSimple {
    private int _number;                          ❶
    private Map<Integer, Integer> _cache;         ❷

    public ImpNumberClassifierSimple(int targetNumber) {
      _number = targetNumber;
      _cache = new HashMap<>();
    }

    public boolean isFactor(int potential) {
      return _number % potential == 0;
    }

    public Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
```

```
        factors.add(_number);
        for (int i = 2; i < _number; i++)
            if (isFactor(i))
                factors.add(i);
        return factors;
    }

    public int aliquotSum() {                          ❸
        if (_cache.get(_number) == null) {
            int sum = 0;
            for (int i : getFactors())
                sum += i;
            _cache.put(_number, sum - _number);
        }
        return _cache.get(_number);
    }

    public boolean isPerfect() {
        return aliquotSum() == _number;
    }

    public boolean isAbundant() {
        return aliquotSum() > _number;
    }

    public boolean isDeficient() {
        return aliquotSum() < _number;
    }
}
```

❶ 内部状态，存放待分类的目标数字。

❷ 内部缓存，防止重复进行不必要的求和运算。

❸ 计算“真约数和”`aliquotSum`，即正约数之和减去数字本身。

例 2-10 中的 `ImpNumberClassifierSimple` 类维持着两个内部状态。其中 `number` 字段的作用是为一系列函数省下一个参数。`cache` 则通过一个 `Map` 结构来缓存每个数字的真约数和，以在后续针对同一个数字的调用中更快地返回结果（查表速度与计算速度的差别）。内部状态在面向对象编程的世界里是受到推崇的平常做法，因为封装被 OOP 语言视为一项优势。状态的划分往往为一些工程实践提供了便利，比如单元测试的时候我们很容易注入各种取值。

例 2-10 的代码经过了精心的组织，划分成很多个小方法。这是测试驱动开发的副产物，不过也正好把算法的各个组成部分都表现了出来。其中一些部分会在后续的改造中逐渐被替换成更加函数式的写法。

2.2.2 稍微向函数式靠拢的完美数分类解法

例 2-10 用它的代码组织形态反映了可测试性的编程目标。假如我们还希望加上一个“最小

化共享状态”的目标，该怎么做呢？这时可以去掉类的成员变量，改为通过参数来传递需要的值。修改后的版本见例 2-11。

例 2-11　稍微向函数式靠拢的完美数分类实现

```
import java.util.Collection;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

public class NumberClassifier {

    public static boolean isFactor(final int candidate, final int number) { ❶
        return number % candidate == 0;
    }

    public static Set<Integer> factors(final int number) {                ❷
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < number; i++)
            if (isFactor(i, number))
                factors.add(i);
        return factors;
    }

    public static int aliquotSum(final Collection<Integer> factors) {      ❸
        int sum = 0;
        int targetNumber = Collections.max(factors);
        for (int n : factors) {
            sum += n;
        }
        return sum - targetNumber;
    }

    public static boolean isPerfect(final int number) {
        return aliquotSum(factors(number)) == number;
    }
                                                                          ❹
    public static boolean isAbundant(final int number) {
        return aliquotSum(factors(number)) > number;
    }

    public static boolean isDeficient(final int number) {
        return aliquotSum(factors(number)) < number;
    }
}
```

❶ 众多方法都必须加上 `number` 参数，因为没有可以存放它的内部状态。

❷ 所有方法都带 `public static` 修饰，因为它们都是纯函数，并因此可以在完美数分类之外的领域使用。

❸ 注意例中对参数类型的选取，尽可能宽泛的参数类型可以增加函数重用的机会。

❹ 例子目前在重复执行分类操作的时候效率较低，因为没有缓存。

在例 2-11 稍微向函数式风格靠拢的 `NumberClassifier` 里面，所有方法都是自足的、带 `public` 和 `static` 作用域的纯函数（即没有副作用的函数）。而由于类里面根本不存在任何内部状态，也就没有理由去“隐藏”任何一个方法。实际上，`factors` 方法在很多其他应用中都有潜在的用途，比如用来寻找素数。

一般来说，面向对象系统里粒度最小的重用单元是类，开发者往往忘记了重用可以在更小的单元上发生。例如，例 2-11 的 `aliquotSum` 方法的参数类型没有选择某一种具体的列表类型，而是定为 `Collection<Integer>`。一个兼容于所有数字集合的接口，在函数级别上发生重用的可能性自然更大一些。

这一版的实现没有为求和结果设计缓存机制。缓存意味着持续存在的状态，可是这一版的实现根本没有可以放置状态的地方。例 2-11 对比例 2-10 相同功能的实现，效率上要低一些。这是因为失去了存放求和结果的内部状态，只好每次都重新计算。我们将在第 4 章借助“记忆”机制，在保持无状态的前提下，把缓存找回来。现在暂且和它告别吧。

2.2.3　完美数分类的Java 8实现

lambda 块是最令 Java 8 面目一新的改进，它其实就是高阶函数。多了这么一个小功能，传统函数式语言里的一些高层次抽象就一下子向 Java 开发者敞开了大门。

请看例 2-12 所示的 Java 8 版完美数分类实现。

例 2-12　完美数分类的 Java 8 实现

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

import static java.lang.Math.sqrt;
import static java.util.stream.Collectors.toList;
import static java.util.stream.IntStream.range;

public class NumberClassifier {

    public static IntStream factorsOf(int number) {
        return range(1, number + 1)
                .filter(potential -> number % potential == 0);
    }

    public static int aliquotSum(int number) {
        return factorsOf(number).sum() - number;
    }

    public static boolean isPerfect(int number) {
```

```
        return aliquotSum(number) == number;
    }

    public static boolean isAbundant(int number) {
        return aliquotSum(number)> number;
    }

    public static boolean isDeficient(int number) {
        return aliquotSum(number) < number;
    }
}
```

例 2-12 的代码明显比原来的命令式解法（例 2-10）以及不完全的函数式版本（例 2-11）短得多，也简单得多。在例 2-12 里，`factorsOf()` 方法返回了一个 `IntStream`，为我们后续串连其他操作，包括令 stream 产生数值输出的终结操作提供了方便。换言之，`factorsOf()` 没有直接返回一个整数列表，而是给了我们一个尚未产生任何输出的 stream。`aliquotSum()` 方法很好写，无非是对约数的列表求和，再减去数本身。我们不需要在例 2-12 中自行编写求和用的 `sum()` 方法，因为 Java 8 已经为我们的 stream 准备了这样一个产生输出的终结操作。

物理上把机械能分成储蓄起来的势能和释放出来的动能。在版本 8 以前的 Java，以及它所代表的许多语言里，集合的行为可以比作动能：各种操作都立即求得结果，不存在中间状态。函数式语言里的 stream 则更像势能，它的操作可以引而不发。被 stream 储蓄起来的有数据来源（例中的数据来源是 `range()` 方法），还有我们对数据设置的各种条件，如例中的筛选操作。只有当程序员通过 `forEach()`、`sum()` 终结操作来向 stream“要”求值结果的时候，才触发从“势能”到“动能”的转换。在“动能”开始释放之前，stream 可以作为参数传递并后续附加更多的条件，继续积蓄它的“势能”。这里关于“势能”的比喻，用函数式编程的说法叫作*缓求值*（lazy evaluation），我们将在第 4 章详细讨论。

旧版本的 Java 语言也有可能写出与例 2-12 风格类似的代码，不过需要克服一点困难，需要用一些框架辅助才行。

2.2.4　完美数分类的Functional Java实现

现在高阶函数已经是新一代语言的标准配备，不过仍然有众多组织因为技术之外的原因，在未来的很多年里都无法摆脱旧版本的 Java 运行时。开源框架 Functional Java 针对 1.5 以上版本的 Java 运行时，以尽可能低的侵入性为代价引入了尽量多的函数式编程手法。例如 Functional Java 可以通过泛型和匿名内部类，在 Java 1.5 时代的 JDK 上模拟出它所缺少的高阶函数特性。例 2-13 是借助 Functional Java 的惯用法来实现的完美数分类，它看上去又和前面的例子有所不同。

例 2-13 使用 Functional Java 框架实现的完美数分类

```
import fj.F;
import fj.data.List;
import static fj.data.List.range;

public class NumberClassifier {

    public List<Integer> factorsOf(final int number) {
        return range(1, number + 1)                                  ❶
                .filter(new F<Integer, Boolean>() {
                    public Boolean f(final Integer i) {
                        return number % i == 0;
                    }
                });                                                  ❷
    }

    public int aliquotSum(List<Integer> factors) {                   ❸
        return factors.foldLeft(fj.function.Integers.add, 0) - factors.last();
    }

    public boolean isPerfect(int number) {
        return aliquotSum(factorsOf(number)) == number;
    }

    public boolean isAbundant(int number) {
        return aliquotSum(factorsOf(number)) > number;
    }

    public boolean isDeficient(int number) {
        return aliquotSum(factorsOf(number)) < number;
    }
}
```

❶ Functional Java 的 `range()` 函数圈出来的是一个左闭右开区间。

❷ 筛选操作代替了迭代。

❸ 折叠（fold）操作代替了迭代。

例 2-13 与例 2-11 的主要区别表现在 `aliquotSum()` 和 `factorsOf()` 这两个方法上。Functional Java 在其 `List` 类中提供的 `foldLeft()` 方法为 `aliquotSum()` 提供了很大的便利。在这个例子里，“fold left”（即左折叠操作）的含义是：

(1) 用一个操作（或者叫运算）将初始值（例中为 0）与列表中的第一个元素结合；
(2) 继续用同样的操作将第 1 步的运算结果与下一个元素结合；
(3) 反复进行直到消耗完列表中的元素。

这几个步骤正好就是我们对数字列表求和的一般做法：从 0 开始，先和第一个元素相加，结果再和第二个元素相加，以此类推直到列表结尾。Functional Java 提供了运算所需的高阶函数（例中的 `Integers.add` 函数），也由它负责施用。当然，真正的高阶函数要到 Java

8 才出现，Functional Java 也无法在旧版本的 Java 里实现完整的高阶函数功能，只是用匿名内部类来模拟高阶函数的编程风格。

例 2-13 另一个值得注意的地方是 `factorsOf()` 方法，它很好地体现了“多着眼结果，少纠结步骤”的格言。寻找一个数的约数，这个问题的实质是什么？或者可以换一种方式来叙述：在从 1 到目标数字的整数列表里，我们怎么确定其中哪些数字是目标数的约数？这样一来，筛选操作就呼之欲出了——我们可以逐一筛选列表中的元素，去除那些不满足筛选条件的数字。`factorsOf()` 方法的作为基本上可以用一句话来描述：对于从 1 到目标数字的区间（不包含区间的右侧端点，因此代码中将区间上限写成 `number + 1`），以 `f()` 方法中的代码来筛选区间内数字所构成的一个列表，`F` 类和 `f()` 方法是 Functional Java 留给我们“填空”数据类型和返回值的地方。

例 2-13 使用了 `foldLeft()` 方法，它依次向左方，即向着第一个元素合并列表。对于满足交换律的加法来说，折叠的方向并不影响结果。万一我们需要使用某些结果与折叠次序相关的操作，还有 `foldRight()` 方法可供选择。

高阶函数消除了摩擦。

你可能会认为 Functional Java 版本（例 2-13）与 Java 8 版本（例 2-12）的区别无非是一些语法糖衣（其实不止）。可是语法上的便利也是很重要的方面，毕竟我们想表达的意思都要由语法来承载。

我跟 Martin Fowler 在巴塞罗那的一辆出租车上有过一次记忆深刻的讨论，我们聊的是 Smalltalk 的衰落和 Java 的兴盛。Fowler 在这两种语言上都有很深厚的积累，他说，起初他觉得从 Smalltalk 到 Java 的变化只是一些语法上的不便，结果却发现被阻碍的还有原先语言所承载的思维方式。在语法处处掣肘下塑造出来的抽象，很难配合我们的思维过程而不产生无谓的摩擦。

不要增加无谓的摩擦。

2.3 具有普遍意义的基本构造单元

我们在举例命令式解法的时候提到了“三板斧”，纵观上面一系列完美数分类的函数式实

现，它们一个不少地出现在每一种实现当中，只是叫法不太一样。在函数式语言和框架里面，这几“板斧”是无处不在的。

2.3.1　筛选

筛选（filter）是列表的一种基本操作：根据用户定义的条件来筛选列表中的条目，并由此产生一个较小的新列表。筛选操作如图 2-1 所示。

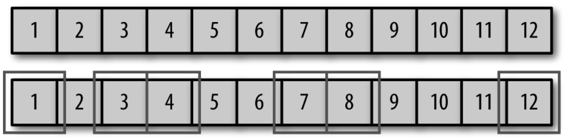


图 2-1：从较大的列表中筛选出一个数字列表

筛选会产生一个新的列表（或集合），其大小根据筛选条件，可能小于原列表。在完美数分类的例子里，我们用了筛选操作来得出数字的约数，如例 2-14 所示。

例 2-14　Java 8 的筛选操作

```
public static IntStream factorsOf(int number) {
    return range(1, number + 1)
            .filter(potential -> number % potential == 0);
}
```

例 2-14 中的代码首先制造一个从 1 到目标数字的区间，然后在该区间上施加 `filter()` 方法，剔除不是目标数约数的数字：Java 的取模运算（`%`）返回整数除法的余数，余数为 0 即表示除数是被除数的约数。

虽然不借助 lambda 块也可以得到相同的结果（如例 2-13），但有的话写起来会简洁很多。例 2-15 是 Groovy 的版本。

例 2-15　Groovy 的筛选操作（叫作 `findAll()`）

```
static def factors(number) {
  (1..number).findAll {number % it == 0}
}
```

例 2-15 省略了参数传递，因为可以直接在闭包内使用单参数占位符，即 `it` 关键字来代表传入的参数；又省略了返回语句，因为方法的最后一行就是方法的返回值，例中恰为约数的列表。

需要根据筛选条件来产生一个子集合的时候，用 `filter`。

2.3.2　映射

映射（map）操作对原集合的每一个元素执行给定的函数，从而变换成一个新的集合，如图 2-2 所示。

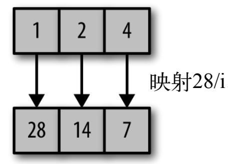


图 2-2：在集合上映射一个函数

为了演示 map() 和相关变换的用法，我对完美数分类的例子做了一点性能上的优化。首先，我创建了一个命令式的版本，如例 2-16 所示。

例 2-16　优化了的完美数分类实现

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import static java.lang.Math.sqrt;

public class ImpNumberClassifier {
    private int _number;                          ❶
    private Map<Integer, Integer> _cache;         ❷

    public ImpNumberClassifier(int targetNumber) {
        _number = targetNumber;
        _cache = new HashMap<>();
    }

    private boolean isFactor(int candidate) {
        return _number % candidate == 0;
    }

    private Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(_number);
        for (int i = 2; i <= sqrt(_number); i++)  ❸
            if (isFactor(i)) {
                factors.add(i);
                factors.add(_number / i);
            }
        return factors;
    }

    private int aliquotSum() {
```

```
        int sum = 0;
        for (int i : getFactors())
            sum += i;
        return sum - _number;
    }

    private int cachedAliquotSum() {                        ❹
        if (_cache.containsKey(_number))
            return _cache.get(_number);
        else {
            int sum = aliquotSum();
            _cache.put(_number, sum);
            return sum;
        }
    }

    public boolean isPerfect() {
        return cachedAliquotSum() == _number;
    }

    public boolean isAbundant() {
        return cachedAliquotSum() > _number;
    }

    public boolean isDeficient() {
        return cachedAliquotSum() < _number;
    }
}
```

❶ 用来放置目标数的内部状态，免得总要在参数里传来传去。

❷ 内部缓存，提高求和结果的查找效率。

❸ `getFactors()` 方法内有一处提高性能的算法优化。优化基于这样的事实：约数总是成对出现的。例如对于数字 16，我们找到约数 2 的时候，也就同时找到了约数 8，因为 2 × 8 = 16。假如我们成对地采集约数，那么只要检查小于或等于目标数平方根的数就可以了。`getFactors()` 方法就是这么做的。

❹ 优先返回缓存的真约数和。

Groovy 当然包含了函数式语言必备的变换函数；它没有用 `map()` 的名字，而是叫作 `collect()`，请看例 2-17。

例 2-17　Groovy 版的约数查找优化算法

```
static def factors(number) {
  def factors = (1..round(sqrt(number)+1)).findAll({number % it == 0})
  (factors + factors.collect {number / it}).unique()
}
```

例 2-17 最后调用了 `unique()` 方法来消除列表中的重复项，确保完全平方数的平方根（如 16 的平方根 4）不会在列表中出现两次。如果想体会一下函数式编程能够将代码改造到什

么地步，请看例 2-18 完美数分类的 Clojure 语言实现。

例 2-18　Clojure 写成的 `(classify )` 函数将所有行为封装在了几行赋值语句里

```
(defn classify [num]
  (let [factors (->> (range 1 (inc num))          ; ❶
                (filter #(zero? (rem num %))))    ; ❷
                sum (reduce + factors)            ; ❸
                aliquot-sum (- sum num)]          ; ❹
    (cond                                         ; ❺
     (= aliquot-sum num) :perfect
     (> aliquot-sum num) :abundant
     (< aliquot-sum num) :deficient)))
```

❶ 方法成了赋值语句。

❷ 把筛选过的区间赋给约数列表。

❸ 把化约（reduce）过的约数列表赋给 `sum`。

❹ 计算真约数和。

❺ 返回代表分类结果的关键字（枚举）。

如果我们让每个函数都合并成一行，那么一系列的函数定义就可以变成一个赋值语句的列表，这就是例 2-18 的真相。Clojure 的 `(let [])` 块允许创建一系列作用于仅限于块内的赋值。首先要计算的是目标数的约数，为此需准备从 1 到目标数的区间 `(range 1 (inc num))`，其中右端点写成 `(inc num)` 是因为 Clojure 的区间定义不包括右端点。接着用 `(filter )` 方法消去不需要的集合元素。一般来说，上述语句按照 Clojure 的习惯写出来应该是 `(filter #(zero? (rem num %)) (range 1 (inc num)))`，不过既然概念上是先有区间再做筛选，那么让代码的阅读次序和思路保持一致会更好一些。Clojure 的 thread-last 宏（即例 2-18 中出现的 `->>` 运算符）可以帮我们做这样的次序调整。求得了全部约数之后，就是对 `sum` 和 `aliquot-sum` 的赋值。函数余下部分的工作是逐条判断 `aliquot-sum` 满足哪一则条件，并返回相应的关键字（以冒号开头的符号，可以当作枚举来使用）。

需要就地变换一个集合的时候，用 `map`。

2.3.3　折叠/化约

第三种基本套路的函数名称最为多样，而且在几种流行语言里的实现各有微妙的区别。`foldLeft` 和 `reduce` 都是 catamorphism 这种范畴论的态射概念具体应用到列表操纵上面的变体，catamorphism 是对列表“折叠”（fold）概念的推广。

reduce 和 fold 操作在功能上大致重合，但根据具体的编程语言而有微妙的区别。两者都用一个*累积量*（accumulator）来“收集”集合元素。reduce 函数一般在需要为累积量设定一个初始值的时候使用，而 fold 起始的时候累积量是空的。函数在操作集合的时候可以有不同的次序，这点会体现在相应的函数命名上（如 foldLeft 和 foldRight）。这里提到的任何一种操作，都不会改变原集合。

我们在 Functional Java 的例子里已经见识过 foldLeft() 函数。所谓“fold left”的含义是：

用一个二元函数或运算符来结合列表的首元素和累积量的初始值（如果累积量有初始值的话）；

重复上一步直到列表耗尽，此时累积量的取值即为折叠运算的结果。

这个过程恰好就是我们对一个数字列表求和的过程：从 0 开始，加上第一个元素，求得的结果再加上第二个元素，就这样一直进行下去，直到列表元素全部用完。

Functional Java 版的完美数分类例子里面有一个 aliquotSum() 方法，它对筛选出来的全部约数求和，如例 2-19 所示。

例 2-19　Functional Java 提供的 foldLeft() 方法

```
public int aliquotSum(List<Integer> factors) {
    return factors.foldLeft(fj.function.Integers.add, 0) - factors.last();
}
```

例 2-19 的方法体只有区区一行，乍看之下不容易明白它究竟如何施展求和操作，去得到 aliquotSum 的结果。例中的折叠操作可理解为令每一个列表元素依次结合的一次变换，变换的结果是从整个列表累积成单独的一个值。左折叠按照从左到右的次序结合列表元素，由一个初始值开始，把元素一个接一个地累加上去，最后得到一个结果。图 2-3 是折叠操作的示意图。

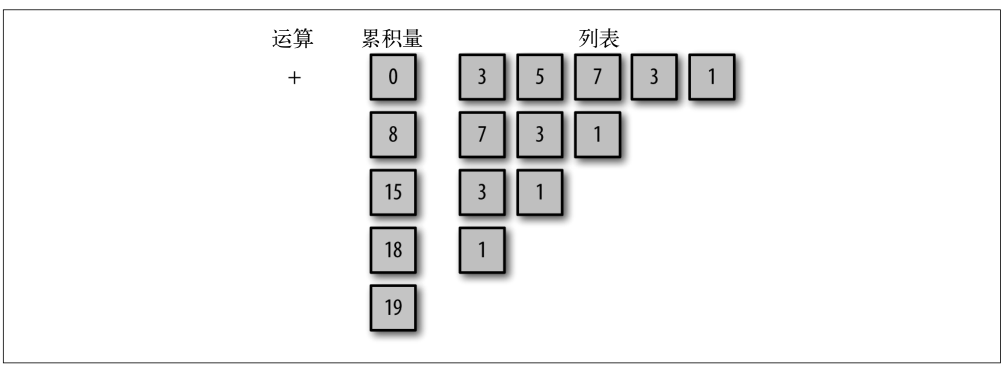


图 2-3：折叠操作

由于加法满足交换律，例 2-19 无论用 `foldLeft()` 还是 `foldRight()` 都将得到同样的结果。但有些运算（包括减法和除法在内）不能随便调换顺序，这时 `foldRight()` 就会派上用场。在纯函数式语言里，左折叠和右折叠的实现并不相同。例如右折叠允许操作无限长度的列表，而左折叠则不允许。

例 2-13 直接使用了 Functional Java 提供的加法运算，别的一些最常用的数学运算也都可以在框架里找到。可是万一我们有更特殊的要求，该怎么做呢？请看例 2-20 的代码。

例 2-20　按用户指定的条件执行 `foldLeft()`

```
static public int addOnlyOddNumbersIn(List<Integer> numbers) {
    return numbers.foldLeft(new F2<Integer, Integer, Integer>() {
        public Integer f(Integer i1, Integer i2) {
            return (!(i2 % 2 == 0)) ? i1 + i2 : i1;
        }
    }, 0);
```

Functional Java 框架专为 Java 8 以前版本的 JDK 而设计，因此不得不创造性地运用单方法接口和匿名内部类来达到目的。内建的 `F2` 类正好具备折叠操作所需的结构，我们用它创建了一个方法，方法的两个参数（将被此方法折叠在一起的两个值）和返回值都为 `Integer` 类型。

化约（reduce）操作在 Groovy 版完美数分类里的用法如例 2-21 所示。

例 2-21　Groovy 版的 `reduce()`（叫作 `inject()`）

```
static def sumFactors(number) {
  factors(number).inject(0, {i, j -> i + j})
}
```

Groovy 的 `inject` 方法与例 2-18 中出现的 `reduce` 函数有着相同的签名；第一个参数都是初始值，第二个参数都是接受两个参数，返回一个值的闭包。例中我们给闭包传的两个参数是 `{i, j → i + j}`。

`fold` 或 `reduce` 常常用在需要从一个集合处理产生另一个大小不同（通常较小但不必然）的集合或单一值的情况。

需要把集合分成一小块一小块来处理的时候，用 `reduce` 或 `fold`。

完美数分类固然是一个做作的例子，很难推广到其他类型的问题。但是我注意到，当项目选用的语言（无论是否函数式语言）支持我们讨论的这些抽象的时候，代码的风格会发生明显的变化。我首先在使用 Ruby on Rails 框架的项目里注意到这种现象。Ruby 语言自然

是支持闭包和 collect()、map()、inject() 等列表操纵方法的，只是它们在代码中出现之频繁令人惊讶。一旦习惯了工具箱里有这样的利器，你就总是会不自觉地拿起它们。

学习函数式编程，或者任何一种新范式都有一个很大的挑战，那就是在掌握新的构造单元之后，还要善于从问题里“发现”它们的身影，从而抓住解答的脉络。函数式编程不会用很多抽象，但每个抽象的泛化程度都很高（特化的方面通过高阶函数注入）。函数式编程以参数传递和函数的复合作为主要的表现手段，我们不需要掌握太多作为“不确定因素”存在的其他语言构造之间的交互规则，这一点对于我们的学习是有利的。

2.4　函数的同义异名问题

作为函数式编程语言的共同特征，我们可以在每一种语言里找到同样的几大类基本函数。然而当开发者从一种语言换到另一种的时候往往不太顺利，原因就是熟悉的函数突然换了一个不认识的名字。继承函数式传统的语言喜欢按照范式术语来命名基本函数，而出自脚本语言背景的则更喜欢使用描述性的名字（有时候还会起多个名字，实质是指向相同函数的别名）。

2.4.1　筛选

筛选函数将用户（通常以高阶函数的形式）给定的布尔逻辑作用于集合，返回由原集合中符合条件的元素组成的一个子集。筛选操作与查找（find）函数的关系很密切，查找函数返回的是集合中第一个符合条件的元素。

1. Scala

Scala 提供了好几种形式的筛选。最简单的一种是在列表上按传入的条件进行筛选。下面的例子首先创建一个数字列表，然后对列表使用 filter() 函数，并在传给函数的代码块中设置筛选条件为可被 3 整除的元素：

```
val numbers = List.range(1, 11)
numbers filter (x => x % 3 == 0)
// List(3, 6, 9)
```

利用 Scala 的隐式参数（implicit parameter）特性可以让例子变得更简短：

```
numbers filter (_ % 3 == 0)
// List(3, 6, 9)
```

第二种写法更精炼，这要归功于 Scala 允许用下划线符号来替换参数。两种写法的执行结果是一样的。

很多筛选操作的例子都用数字来演示，其实 filter() 可以用于任意的集合。下面的例子在元素为单词的集合上使用 filter() 函数来找出由 3 个字母构成的单词：

```
val words = List("the", "quick", "brown", "fox", "jumped",
                           "over", "the", "lazy", "dog")
words filter (_.length == 3)
// List(the, fox, the, dog)
```

Scala 的第二种筛选形式是 `partition()` 函数，其返回结果是由原集合的内容划分而成的两个集合，原集合本身保持不变。划分的依据是用户传进来作为筛选条件的高阶函数。下面的例子以能否被 3 整除为标准，用 `partition()` 函数把数字列表分成了两部分：

```
numbers partition (_ % 3 == 0)
// (List(3, 6, 9),List(1, 2, 4, 5, 7, 8, 10))
```

`filter()` 函数返回所有匹配元素的集合，而 `find()` 只返回第一个匹配项：

```
numbers find (_ % 3 == 0)
// Some(3)
```

不过，`find()` 并不直接把匹配项作为返回值，而是 `Option` 类作了一层包装。`Option` 有两个可能的取值：`Some` 或者 `None`。Scala 也像别的函数式语言一样，用 `Option` 来作为一种迂回手段，以避免在无返回值的情况下返回 `null`。真正的返回值包裹在 `Some()` 实例之中，对于 `numbers find (_ % 3 == 0)` 来说，这个值是 `3`。如果要查找的内容不存在，那么返回的就是 `None` 了：

```
numbers find (_ < 0)
// None
```

我们将在第 5 章继续深入探讨 `Option` 以及其他功能类似的类。

Scala 还有若干处理集合的函数，也是根据一个传入的断言来决定元素去留的。`takeWhile()` 函数从集合头部开始，一直取到第一个不满足断言的元素：

```
List(1, 2, 3, -4, 5, 6, 7, 8, 9, 10) takeWhile (_ > 0)
// List(1, 2, 3)
```

`dropWhile()` 函数则从集合头部开始，一直丢弃满足断言的元素，直到遇到第一个非匹配项：

```
words dropWhile (_ startsWith "t")
// List(quick, brown, fox, jumped, over, the, lazy, dog)
```

2. Groovy

Groovy 一般不被看作一种函数式语言，但它具备很多函数式的范式，只是命名上往往带有脚本语言的色彩。例如按照函数式语言的传统一般叫作 `filter()` 的函数，对应的是 Groovy 的 `findAll()` 方法：

```
(1..10).findAll {it % 3 == 0}
// [3, 6, 9]
```

这个方法也像 Scala 的筛选函数一样，适用于所有的类型，包括字符串：

```
def words = ["the", "quick", "brown", "fox", "jumped",
                     "over", "the", "lazy", "dog"]
words.findAll {it.length() == 3}
// [The, fox, the, dog]
```

Groovy 也有跟 `partition()` 对应的函数，叫作 `split()`：

```
(1..10).split {it % 3}
// [ [1, 2, 4, 5, 7, 8, 10], [3, 6, 9] ]
```

`split()` 方法的返回值是一个嵌套的数组，类似于 Scala 的 `partition()` 函数返回的嵌套列表。

Groovy 的 `find()` 方法返回集合中的第一个匹配项：

```
(1..10).find {it % 3 == 0}
// 3
```

当 `find()` 找不到匹配项的时候，Groovy 没有采用 Scala 防范空值的做法，而是按照 Java 的习惯直接返回 `null`。

```
(1..10).find {it < 0}
// null
```

Groovy 也有 `takeWhile()` 和 `dropWhile()` 方法，其语义和 Scala 的版本差不多：

```
[1, 2, 3, -4, 5, 6, 7, 8, 9, 10].takeWhile {it > 0}
// [1, 2, 3]

words.dropWhile {it.startsWith("t")}
// [quick, brown, fox, jumped, over, the, lazy, dog]
```

和 Scala 的例子一样，Groovy 的 `dropWhile()` 也是作为一种特殊的筛选来使用的。它丢弃满足断言的最长前缀，换言之，被筛选到的只是列表开头的一部分：

```
def moreWords = ["the", "two", "ton"] + words
moreWords.dropWhile {it.startsWith("t")}
// [quick, brown, fox, jumped, over, the, lazy, dog]
```

3. Clojure

Clojure 用于操纵集合的招式数量多得惊人，而且因为 Clojure 语言的动态类型特征，这些函数一般还都是泛型的函数。很多倾心于 Clojure 的开发者正是被它丰富而灵活的集合库所吸引。Clojure 在命名上沿袭函数式编程的传统，这一点可以在 `(filter )` 函数的名字里看出来：

```
(def numbers (range 1 11))
(filter (fn [x] (= 0 (rem x 3))) numbers)
; (3 6 9)
```

Clojure 和另外两种语言一样，提供了针对简单匿名函数的简写语法：

```
(filter #(zero? (rem % 3)) numbers)
; (3 6 9)
```

Clojure 的函数也像另外两种语言一样，适用于各种类型，包括字符串：

```
(def words ["the" "quick" "brown" "fox" "jumped" "over" "the" "lazy" "dog"])
(filter #(= 3 (count %)) words)
; (the fox the dog)
```

Clojure 给 (`filter`) 设定的返回值类型是 `Seq`。`Seq` 接口是 Clojure 用于表示序列型集合的核心抽象，用一对圆括号括起来的就是一个 `Seq`。

2.4.2 映射

第二种存在于所有函数式语言中的主要的函数式变换是映射。传给映射函数的是一个高阶函数和一个集合，它在对集合中的每一个元素施用传入的函数之后，产生另一个集合作为返回值。返回的集合大小与原来传入的集合相同（这一点不同于筛选操作），只是元素的取值变了。

1. Scala

Scala 的 `map()` 函数接受一个代码块作为参数并返回变换后的集合：

```
List(1, 2, 3, 4, 5) map (_ + 1)
// List(2, 3, 4, 5, 6)
```

`map()` 函数适用于各种元素类型的集合，不过变换后的集合元素不一定还是原来的类型。例如下面的代码返回了一个由原字符串集合中每个元素的长度组成的列表：

```
words map (_.length)
// List(3, 5, 5, 3, 6, 4, 3, 4, 3)
```

嵌套的列表在函数式编程语言中运用得极为频繁，因此各语言普遍地具备用来消除嵌套的库函数，一般将此操作称为“展平”（flattening）。下面的例子对一个嵌套的列表执行展平的操作：

```
List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9)) flatMap (_.toList)
// List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

展平后得到一个去掉了额外的数据结构，只保留所有元素本身的列表。`flatMap()` 函数还可以用在一些在传统眼光看来不存在嵌套的数据结构上。例如我们可以把字符串看成一系

列嵌套在一起的字符：

```
words flatMap (_.toList)
// List(t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x, ...
```

2. Groovy

映射操作在 Groovy 语言里对应的是若干 collect() 函数。其中的基本形式以一个代码块为参数，并将之施用到集合中的每个元素：

```
(1..5).collect {it += 1}
// [2, 3, 4, 5, 6]
```

Groovy 和另外两种语言一样，提供了针对简单匿名高阶函数的简写语法。它用 it 关键字作为参数占位标记。

只要配上合适的断言（也就是返回值为 true 或 false 的函数），collect() 方法可以用在任意的集合上。用来处理字符串列表自然不成问题：

```
def words = ["the", "quick", "brown", "fox", "jumped",
                     "over", "the", "lazy", "dog"]
words.collect {it.length()}
// [3, 5, 5, 3, 6, 4, 3, 4, 3]
```

Groovy 也有一个类似于 flatMap()，用来消除嵌套结构的方法，叫作 flatten()：

```
[ [1, 2, 3], [4, 5, 6], [7, 8, 9] ].flatten()
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

flatten() 方法同样适用于一些非典型的集合，如字符串：

```
(words.collect {it.toList()}).flatten()
// [t, h, e, q, u, i, c, k, b, r, o, w, n, f, o, x, j, ...
```

3. Clojure

Clojure 有 (map) 函数，其参数为一个高阶函数（包括各种运算符在内）和一个集合：

```
(map inc numbers)
; (2 3 4 5 6 7 8 9 10 11)
```

(map) 的第一个参数可以是任意的单参数函数，无论命名函数、匿名函数都可以，内建函数也包括在内，如例中对参数进行递增的 inc。下面的代码根据字符串中每个单词的长度生成了一个列表，例中使用了典型的匿名语法：

```
(map #(count %) words)
; (3 5 5 3 6 4 3 4 3)
```

Clojure 的 (flatten) 函数类似于 Groovy：

```
(flatten [[1 2 3] [4 5 6] [7 8 9]])
; (1 2 3 4 5 6 7 8 9)
```

2.4.3 折叠/化约

第三种常用函数在名称上变化最多，而且不同实现之间有着诸多微妙差异。

1. Scala

Scala 的各种折叠操作最为丰富，其中部分原因是它需要更多地面对某些类型相关的场景，而这些场景在动态类型的 Groovy 和 Clojure 语言中根本不存在。化约操作通常用来完成求和的工作：

```
List.range(1, 10) reduceLeft((a, b) => a + b)
// 45
```

传给 `reduce()` 的函数或运算符一般接受两个参数，且仅返回单一值，就好像原集合被“消耗”掉了一样。利用 Scala 提供的语法糖衣可以让函数定义变得简短一些：

```
List.range(1, 10).reduceLeft(0)(_ + _)
// 45
```

`reduceLeft()` 函数假定集合的第一个元素是运算的左值。对于加法这样的运算来说，操作数的摆放不影响运算的结果，但对于除法这样的运算来说，运算次序是决定性的。如果我们希望调转运算进行的方向，可以改用 `reduceRight()`：

```
List.range(1, 10) reduceRight(_ - _)

// 8 - 9 = -1
// 7 - (-1) = 8
// 6 - 8 = -2
// 5 - (-2) = 7
// 4 - 7 = -3
// 3 - (-3) = 6
// 2 - 6 = -4
// 1 - (-4) = 5
// result: 5
```

这里所谓“调转方向”的实际意义可能不太直观。`reduceRight()` 调转的是运算的方向，而不是参数的次序。因此，它首先计算 `8 - 9`，得到的结果再作为第二个参数参与后续的计算。

懂得什么时候应该使用像“reduce”这样的高层次抽象，是掌握函数式编程的一处关键所在。下面的例子用了 `reduceLeft()` 来找出集合中最长的单词：

```
words.reduceLeft((a, b) => if (a.length > b.length) a else b)
// jumped
```

化约操作和折叠操作在功能上存在交集，其中的微妙差异前文已经讨论过。而就在它们的共同用途上，两者也有一处明显的区别。按照 Scala 的定义，`reduceLeft[B >: A](op: (B, A) => B): B` 的方法签名表明它只要求提供一个参数，即用来结合集合中元素的函数。起始值被指定为集合的第一个元素，不必另外提供。相比之下，方法签名 `foldLeft[B](z: B)(op: (B, A) => B): B` 就要求提供一个起始值来作为后续计算的种子，这个另外提供的值同时也意味着返回值的类型可以不同于列表元素的类型。

下面的例子用 `foldLeft()` 来对一个集合求和：

```
List.range(1, 10).foldLeft(0)(_ + _)
// 45
```

Scala 语言支持运算符重载，`foldLeft` 和 `foldRight` 作为十分常用的折叠操作，也分别有各自对应的运算符 `/:` 和 `:\`。于是上面的求和例子可以写得更简短一些：

```
(0 /: List.range(1, 10)) (_ + _)
// 45
```

类似地，如果想计算列表的累减结果（即累加求和的逆运算，虽然这种需求很少见），我们可以用 `foldRight()` 函数，也可以用 `:\` 运算符：

```
(List.range(1, 10) :\ 0) (_ - _)
// 5
```

2. Groovy

Groovy 提供了两个版本的 `inject()` 来完成化约操作，分别对应于 Scala 众多同类方法中的 `reduce()` 和 `foldLeft()`。其中一个版本的 `inject()` 允许传入初始值。下面的例子使用 `inject()` 方法来对集合中的元素求和：

```
(1..10).inject {a, b -> a + b}
// 55
```

也可以使用带初始值的版本：

```
(1..10).inject(0, {a, b -> a + b})
// 55
```

Groovy 的函数式类库远不如 Scala 和 Clojure 丰富。这一点并不奇怪，毕竟 Groovy 的定位是一种多范式语言，并不特别强调函数式编程能力。

3. Clojure

Clojure 的基本定位就是一种函数式编程语言，所以它肯定支持 `(reduce )`。`(reduce )` 函数有一个可选的初始值参数，因此它其实涵盖了 Scala 中的 `reduce()` 和 `foldLeft()` 两种情况。`(reduce )` 函数该是什么样子，我们都已经很熟悉了。它要求传入一个双参数的函数

和一个集合：

```
(reduce + (range 1 11))
; 55
```

Clojure 在它的 Reducers 库（http://clojure.org/reducers）里提供了更多与化约操作相关的高级功能。

学习新范式（如函数式编程）的困难有一部分在于学习新的术语。假如遇到不同社群使用不同术语的情况，想搞清楚就更困难了。不过只要你抓住不同语言的共同点去观察，就能够看穿在形形色色的语法遮挡之下，其实功能大同小异。

第3章

权责让渡

坦白说，我再也不想自找罪受去用一种没有垃圾收集的语言。经历过 C++ 和其他同时代语言那么多年的煎熬，我是一点都不愿意拱手交出现代语言带来的便利。软件开发的进步过程就是这样。我们构造一层又一层的抽象来处理（并隐藏）琐碎的细节。随着硬件能力的提高，我们将越来越多的任务转嫁给语言和运行时。开发者曾经因为速度太慢而排斥解释型语言，现在它们已经随处可见。函数式语言的很多特性十年前还慢得叫人提不起一点兴趣，现在却已经成了节约开发者时间和精力的灵丹妙药。

函数式思维的好处之一，是能够将低层次细节（如垃圾收集）的控制权移交给运行时，从而消弭了一大批注定会发生的程序错误。开发者们可以一边熟视无睹地享受着最基本的抽象，比如内存，一边却会对更高层次的抽象感觉突兀。然而不管层次高低，抽象的目的总是一样的：让开发者从繁琐的运作细节里解脱出来，去解答问题中非重复性的那些方面。

本章将展示在函数式语言中，向语言和运行时让渡控制权的五种途径，让开发者抛开负累，投入到更有意义的问题中去。

3.1 迭代让位于高阶函数

其实在上一章已经演示过让出控制权的第一个例子，我们在例 2-3 里面，用 `map` 等函数替换了迭代。这笔“交易”的得失很清楚：如果能够用高阶函数把希望执行的操作表达出来，语言将会把操作安排得更高效，甚至只要增加一行 `par` 修饰，就能够让操作并行化。

多线程代码属于最难编写，最容易出错，也最难调试的类别。只有卸下线程管理这份头痛的差事，开发者才能少一些低层次的琐碎操劳。

这样说并不等于开发者应该抛开所有的责任，不去理解低层次抽象的来龙去脉。在很多情况下，我们使用一个抽象，比如 Stream 的时候，必须清楚可能产生的连带后果。很多开发者都没认识到，即使有了 Java 8 的 Stream API，他们仍然需要理解 Fork/Join 库的细节才能写出高性能的代码。当你掌握了背后的原理，才能把力量用在最正确的地方。

> 理解掌握的抽象层次永远要比日常使用的抽象层次更深一层。

程序员的工作效率依赖于抽象层，好比没有人会直接翻弄硬盘上或 0 或 1 的磁记录来给计算机编程。抽象隐藏了繁杂的细节，只是有时候会连同重要的考虑因素一起隐藏掉。这方面的问题将在第 8 章展开探讨。

3.2 闭包

闭包（closure）是所有函数式语言都具备的一项平常特性，可是相关的论述却常常充斥着晦涩乃至神秘的字眼。所谓闭包，实际上是一种特殊的函数，它在暗地里绑定了函数内部引用的所有变量。换句话说，这种函数（或方法）把它引用的所有东西都放在一个上下文里“包”了起来。

例 3-1 是一个简单的例子，这段 Groovy 代码演示了闭包的创建和绑定。

例 3-1　Groovy 语言中闭包绑定的简单示例

```
class Employee {
    def name, salary
}

def paidMore(amount) {
    return {Employee e -> e.salary > amount}
}

isHighPaid = paidMore(100000)
```

例 3-1 首先定义了一个简单的 Employee 类，类中带有两个字段。接着定义带有 amount 参数的 paidMore 函数，其返回值是一个以 Employee 实例为参数的代码块，或者叫闭包。类型声明 Employee 可写可不写，这里写出来顺便起到文档的作用。接下来，我们给代码块传入参数值 100 000，并赋予 isHighPaid 的名称，于是数值 100 000 就随着这一步赋值操作，永久地和代码块绑定在一起了。以后有员工数据被代入这个代码块求解的时候，它就可以拿绑定的数值作为标准去评判员工的工资高低。

例 3-2　执行闭包

```
def Smithers = new Employee(name:"Fred", salary:120000)
def Homer = new Employee(name:"Homer", salary:80000)
println isHighPaid(Smithers)
println isHighPaid(Homer)
// true, false
```

例 3-2 创建了两笔员工数据，然后判断其工资是否达到标准线。闭包在生成的时候，会把引用的变量全部圈到代码块的作用域里，封闭、包围起来（故名闭包）。闭包的每个实例都保有自己的一份变量取值，包括私有变量也是如此。也就是说，我们可以创建 `paidMore` 闭包的另一个实例，给它绑定另外的数值（当然实例的名字也要另取），如例 3-3 所示。

例 3-3　绑定另一个闭包

```
isHigherPaid = paidMore(200000)
println isHigherPaid(Smithers)
println isHigherPaid(Homer)
def Burns = new Employee(name:"Monty", salary:1000000)
println isHigherPaid(Burns)
// false, false, true
```

闭包经常被函数式语言和框架当作一种异地执行的机制，用来传递待执行的变换代码，如 `map()` 之类的高阶函数。在缺乏闭包特性的旧版 Java 平台上，Functional Java 利用匿名内部类来模仿“真正的”闭包的某些行为，但语言的先天不足导致这种模仿是不彻底的。闭包的执行机制究竟有什么玄机？

我们用一个例子来说明闭包的特殊之处，请看例 3-4。

例 3-4　闭包的原理（Groovy 示例）

```
def Closure makeCounter() {
  def local_variable = 0
  return { return local_variable += 1 } // ❶
}

c1 = makeCounter()    // ❷
c1()                  // ❸
c1()
c1()

c2 = makeCounter()    // ❹

println "C1 = ${c1()}, C2 = ${c2()}"
// output: C1 = 4, C2 = 1  // ❺
```

❶ 函数的返回值是一个代码块，而不是一个值。

❷ `c1` 现在指向代码块的一个实例。

❸ 调用 `c1` 将递增其内部变量，如果这个时候输出，其结果会是 `1`。

❹ `c2` 现在指向 `makeCounter()` 的一个全新实例，与其他实例没有关联。

❺ 每个实例的内部状态都是独立的，各自拥有一份 `local_variable`。

`makeCounter()` 函数首先定义一个局部变量，明白无误地命名为 `local_variable`，接着返回一个使用了该局部变量的代码块。注意 `makeCounter()` 函数的返回类型是 `Closure`，而不是一个单纯的值。代码块的工作仅仅是递增并返回其局部变量的值。方法中两次明确写出了 `return` 关键字，其实这两个地方 Groovy 都允许省略，不过那样的话，代码看起来就有些晦涩了。

为了演示 `makeCounter()` 函数的用法，我们给代码块分配了一个变量名 `c1`，然后调用了三次。调用代码块的时候用到了 Groovy 提供的语法糖衣，也就是在代码块变量名后直接跟一对圆括号的写法（否则应该写成 `c1.call()`）。接下来，我们第二次调用了 `makeCounter()`，将返回的又一个代码块实例赋给变量 `c2`。最后我们把 `c1` 和 `c2` 都调用了一次。从运行的结果可以看出来，两个代码块实例都分别持有自己的一份 `local_variable` 变量。"闭包"这个名字来源于它创建封闭上下文的行为。虽然局部变量不是在代码块里面定义的，但只要代码块引用了该变量，两者就被绑定在一起，这种联系在代码块实例的全部生命期内都一直保持着。

从实现的角度来说，代码块实例从它被创建的一刻起，就持有其作用域内一切事物的封闭副本，如例 3-4 的 `local_variable`。当代码块实例被垃圾收集的时候，它持有的引用也同时被回收。

像例 3-4 那样创建一个闭包仅仅为了修改自身的内部状态，不是值得提倡的闭包用法，我们这样写只是为了演示闭包绑定的运作原理。更常见的用法是绑定常量或者不可变的值（如例 3-1）。

在 Java 8 以前版本的 Java 语言，或者任意一种支持函数而不支持闭包的语言里面，我们最多能模拟到例 3-5 的程度。

例 3-5　Java 版的 `makeCounter()`

```
class Counter {
    public int varField;

    Counter(int var) {
        varField = var;
    }

    public static Counter makeCounter() {
        return new Counter(0);
    }

    public int execute() {
        return ++varField;
    }
}
```

Counter 类还可以有别的一些写法（比如写成匿名的、泛型的，等等），但不管怎么做，都避免不了要自己去管理状态。闭包在这里表现出来的函数式思维就是“让运行时去管理状态”。比起自己硬着头皮去处理字段创建、呵护状态（包括经受多线程环境的严酷考验）这些繁琐的事务，还不如交出对状态的控制权，让语言和框架悄悄在背后帮我们管理好。

让语言去管理状态。

闭包还是*推迟执行*原则的绝佳样板。我们把代码绑定到闭包之后，可以推迟到适当的时机再执行闭包。这个特点在很多场合都能发挥作用。例如必要的变量和函数可能并不在定义时的作用域里，要到执行的时候才准备好。那么我们把执行上下文放在闭包里保留起来，就可以等到正确的时机再完成执行。

命令式语言围绕状态来建立编程模型，参数传递是其典型特征。闭包作为一种*对行为*的建模手段，让我们把代码和上下文同时封装在单一结构，也就是闭包本身里面，像传统数据结构一样可以传递到其他位置，然后在恰当的时间和地点完成执行。

抓住上下文，而非状态。

3.3　柯里化和函数的部分施用

柯里化（currying）和函数的*部分施用*（partial application）都是从数学里借用过来的编程语言技法（基于 20 世纪 Haskell Curry 等数学家的研究成果）。这两种技法以不同的面目出现在各种类型的语言里，在函数式语言当中尤为普遍。柯里化和部分施用都有能力操纵函数或方法的参数数目，一般是通过向一部分参数代入一个或多个默认值的办法来实现的（这部分参数被称为“固定参数”）。大多数函数式语言都具备柯里化和部分施用这两种特性，但实现上各有各的做法。

3.3.1　定义与辨析

乍看起来，柯里化和部分施用的使用效果是一样的。两者都可以创建有一部分预设参数值的函数。

柯里化指的是从一个多参数函数变成一连串单参数函数的变换。它描述的是变换的过程，

不涉及变换之后对函数的调用。调用者可以决定对多少个参数实施变换，余下的部分将衍生为一个参数数目较少的新函数。

部分施用指通过提前代入一部分参数值，使一个多参数函数得以省略部分参数，从而转化为一个参数数目较少的函数。这种技法叫作“部分施用”，顾名思义，就是让函数先作用于其中一些参数，经过部分的求解，结果返回一个由余下参数构成签名的函数。

柯里化和部分施用都是在我们提供部分参数值之后，产出可以凭余下参数实施调用的一个函数。不同的地方在于，函数柯里化的结果是返回链条中的下一个函数，而部分施用是把参数的取值绑定到用户在操作中提供的具体值上，因而产生一个“元数”（参数的数目）较少的函数。用元数大于二的函数来套一下这里的解释，它们之间的区别就会比较清楚了。

举个例子，函数 `process(x, y, z)` 完全柯里化之后将变成 `process(x)(y)(z)` 的形式，其中 `process(x)` 和 `process(x)(y)` 都是单参数的函数。如果只对第一个参数柯里化，那么 `process(x)` 的返回值将是一个单参数的函数，而这个唯一的参数又接受另一个参数的输入。而部分施用的结果直接是一个减少了元数的函数。如果在 `process(x, y, z)` 上部分施用一个参数，那么我们将得到还剩下两个参数的函数：`process(y, z)`。

这两种技法的区分很重要而且很容易被错误地理解，可是使用中它们偏偏又经常得到相同的结果。这里还有更加添乱的事情，Groovy 实现了部分施用也实现了柯里化，但是它把两者都叫作柯里化。Scala 既有部分施用函数（partially applied function），又有名称相近的偏函数类 `PartialFunction`，可它们是截然不同的两个概念。

3.3.2 Groovy的情况

Groovy 通过 `curry()` 函数实现柯里化，这个函数来自 `Closure` 类。

例 3-6 Groovy 语言中的柯里化

```
def product = { x, y -> x * y }

def quadrate = product.curry(4)        ❶
def octate = product.curry(8)          ❷

println "4x4: ${quadrate.call(4)}"     ❸
println "8x5: ${octate(5)}"            ❹
```

❶ 调用 `curry()` 来固定一个参数，返回结果是一个单参数的函数。

❷ `octate()` 函数总是对传入的参数乘以 `8`。

❸ `quadrate()` 是一个单参数的函数，可以通过 `Closure` 类的 `call()` 方法来调用它。

❹ Groovy 提供了一层语法糖衣，可以让调用语句的写法更自然一些。

例 3-6 首先定义接受两个参数的代码块 `product`。我们利用 Groovy 内建的 `curry()` 方法，在 `product` 的基础上构造出两个新的代码块，`quadrate` 和 `octate`。Groovy 为调用代码块提供了特别的便利，我们既可以显式执行 `call()` 方法，也可以使用 Groovy 在语言层面提供的语法糖衣，也就是在代码块的名称后紧跟一对圆括号，参数则写在括号里（如例中 `octate(5)` 的写法）。

`curry()` 虽然叫这个名字，它在背后对代码块所做的事情其实属于函数的部分施用。尽管名不副实，但用它来模拟出柯里化的效果还是可行的，做法是通过连续的部分施用使函数变形为一连串单参数的函数，如例 3-7 所示。

例 3-7　Groovy 语言中部分施用与柯里化的对比

```
def volume = {h, w, l -> h * w * l}
def area = volume.curry(1)
def lengthPA = volume.curry(1, 1)          ❶
def lengthC = volume.curry(1).curry(1)    ❷

println "参数取值为2x3x4的长方体,体积为${volume(2, 3, 4)}"
println "参数取值为3x4的长方形,面积为${area(3, 4)}"
println "参数取值为6的线段,长度为${lengthPA(6)}"
println "参数取值为6的线段,经柯里化函数求得的长度为${lengthC(6)}"
```

❶ 部分施用。

❷ 柯里化。

例 3-7 中 `volume` 代码块的作用是按照公式计算长方体的体积。接着我们固定长方体的第一维（即代表高度的参数 `h`），令其取值为 `1`，从而构造出第二个代码块 `area`（作用是计算长方形的面积）。如果我们继续以 `volume` 为基础构造计算线段长度的代码块，那么无论使用部分施用还是柯里化的技法都能完成任务。`lengthPA` 通过部分施用将前两个参数都固定为 `1`。`lengthC` 连续做了两次柯里化，最后算得与 `lengthPA` 相同的结果。两种写法只有微妙的区别，最终的计算结果也完全相同，但如果你在一名函数式程序员面前不加区分地使用这两个名词，他一定会纠正你。很不幸，Groovy 把这两个密切相关的概念混为一谈了。

函数式编程赋予我们另一套新的构造单元，代替以往命令式语言所使用的机制来完成相同的目标。这些构造单元之间的关系经过了细致的安排。复合（composition），是函数式语言拼组这些构造单元的一般方式，这方面的详细讨论放在第 6 章。请看例 3-8 的 Groovy 代码。

例 3-8　Groovy 语言中函数的复合

```
def composite = { f, g, x -> return f(g(x)) }
def thirtyTwoer = composite.curry(quadrate, octate)

println "composition of curried functions yields ${thirtyTwoer(2)}"
```

例 3-8 定义了一个复合的代码块，由两个函数构成，或者更准确地说，是在一个函数的返回值上调用另一个函数。然后我们利用它来构造 `thirtyTwoer` 代码块，其中运用了部分施用的手法来组合 `quadrate` 和 `octate` 两个函数。

3.3.3　Clojure的情况

Clojure 有一个 `(partial f a1 a2 ...)` 函数，我们传给它函数 `f` 和若干数量不足的参数，它将返回经过部分施用的函数 `f`，可凭余下的参数进行调用。例 3-9 演示了两个例子。

例 3-9　Clojure 语言中的部分施用技法

```
(def subtract-from-hundred (partial - 100))

(subtract-from-hundred 10)      ; same as (- 100 10)
; 90

(subtract-from-hundred 10 20)   ; same as (- 100 10 20)
; 70
```

例 3-9 将 `subtract-from-hundred` 函数定义为部分施用“`-`”运算符（Clojure 语言对运算符和函数进行了区分），并设定了部分施用的参数 `100`。Clojure 的部分施用可以用在单参数函数上，也可以用在多参数函数上，例 3-9 分别给出了例子。

由于 Clojure 是动态类型的语言，并且支持可变长度的参数列表，它没有将柯里化实现成一种语言特性，相关的场景交由部分施用去处理。不过 Clojure 在 Reducers 库里有一个命名空间内私有的 `(defcurried ...)` 函数，虽然其本意是方便库内的函数定义，但凭借 Lisp 家族血脉里与生俱来的灵活性，扩大一下 `(defcurried ...)` 的使用范围简直小菜一碟。

3.3.4　Scala的情况

Scala 支持柯里化和部分施用，另外还有一个用来定义偏函数的 trait。

1. 柯里化

Scala 允许函数定义多组参数列表，每组写在一对圆括号里。当我们用少于定义数目的参数来调用函数的时候，将返回一个以余下的参数列表为参数的函数。请看来自 Scala 文档的例 3-10。

例 3-10　Scala 语言中的参数柯里化

```
def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

def modN(n: Int)(x: Int) = ((x % n) == 0)
```

```
val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
println(filter(nums, modN(2)))
println(filter(nums, modN(3)))
```

例 3-10 中的 `filter()` 函数递归地执行传入的筛选条件。筛选条件 `modN()` 函数定义了两组参数列表，而我们在经 `filter()` 调用它的时候，只传入了一个参数。作为 `modN()` 柯里化的结果，我们得到一个参数为 `Int` 类型并返回 `Boolean` 类型的函数，正好符合 `filter()` 函数定义中对其第二个参数的要求。

2. 部分施用函数

Scala 也支持函数的部分施用，如例 3-11 所示。

例 3-11　Scala 语言中函数的部分施用

```
def price(product : String) : Double =
  product match {
    case "apples" => 140
    case "oranges" => 223
}

def withTax(cost: Double, state: String) : Double =
  state match {
    case "NY" => cost * 2
    case "FL" => cost * 3
}


val locallyTaxed = withTax(_: Double, "NY")
val costOfApples = locallyTaxed(price("apples"))

assert(Math.round(costOfApples) == 280)
```

例 3-11 首先定义了从货品映射到价格的 `price()` 函数。接着又定义了以价格和所属州为参数计算税后价的 `withTax()` 函数。现在请考虑这样一种使用场景，假如我们知道，当前的代码文件只会涉及其中一个州的税率，那么每一次调用 `withTax()` 都要带上相同的州参数就显得很累赘了。这时我们就可以对州参数做部分施用，得到一个固定了州参数值的函数版本。经过这样的处理，`locallyTaxed()` 函数就只需要传给它价格参数了。

3. 偏函数

Scala 设计出 `PartialFunction` trait 是为了密切配合语言中的模式匹配特性，其详情可参阅第 6 章。尽管名称相似，`PartialFunction` trait 并不生成部分施用函数。它的真正用途是描述只对定义域中一部分取值或类型有意义的函数。

Case 语句是偏函数的一种用法。例 3-12 的 Scala 代码单独使用了 `case` 关键字，没有出现习惯上总是和 `case` 搭配在一起的 `match`。

例 3-12　不和 match 一起出现的 case

```
val cities = Map("Atlanta" -> "GA", "New York" -> "New York",
  "Chicago" -> "IL", "San Francsico " -> "CA", "Dallas" -> "TX")

cities map { case (k, v) => println(k + " -> " + v) }
```

例 3-12 先创建了反映城市与所属州对应关系的一个 Map。然后我们在集合上调用 map() 函数，把键值对的内容逐一拆开并打印出来。在 Scala 语言里，含有 case 语句的代码块是匿名函数的一种定义方式。不带上 case，我们可以写出更加简练的匿名函数定义，但 case 语法有着额外的好处，请看例 3-13 的说明。

例 3-13　map 和 collect 的区别

```
List(1, 3, 5, "seven") map { case i: Int => i + 1 } // 无法顺利完成
// scala.MatchError: seven (of class java.lang.String)

List(1, 3, 5, "seven") collect { case i: Int => i + 1 }
// 验证结果
assert(List(2, 4, 6) == (List(1, 3, 5, "seven") collect { case i: Int => i + 1 }))
```

从例 3-13 可以看到，我们无法顺利地在一个混杂不同类型元素的集合上执行带着 case 匿名函数的 map 操作：当函数企图对 "seven" 字符串做算术递增的时候，运行时会给我们一个 MatchError。可是另一行的 collect() 操作就能正确执行完。为什么会有这样的差别？错误怎么不见了呢？

Case 语句定义了偏函数（partial function），请注意不要和名称相近的部分施用函数相混淆。偏函数的参数被限定了取值范围。例如数学函数 1/x 在 x = 0 的时候是无意义的。

偏函数提供了一种对参数取值设置约束条件的途径。例 3-13 在执行 collect() 操作的时候，取值条件是为 Int 而设的，String 类型不包括在内，所以字符串 "seven" 没有被采集。

我们还可以直接使用 PartialFunction trait 来定义偏函数，如例 3-14 所示。

例 3-14　在 Scala 语言中定义偏函数

```
val answerUnits = new PartialFunction[Int, Int] {
    def apply(d: Int) = 42 / d
    def isDefinedAt(d: Int) = d != 0
}

assert(answerUnits.isDefinedAt(42))
assert(! answerUnits.isDefinedAt(0))
assert(answerUnits(42) == 1)
//answerUnits(0)
//java.lang.ArithmeticException: / by zero
```

例 3-14 从 PartialFunction trait 派生出 answerUnits，并实现了两个函数，apply() 和 isDefinedAt()。其中 apply() 函数负责具体的运算。另一个方法 isDefinedAt() 是定义一个 PartialFunction 的硬性要求，我们就在这里设置判断参数是否有效的约束条件。

由于 Scala 允许我们用 `case` 代码块来定义偏函数，例 3-14 的 `answerUnits` 可以改成更加简练的写法，如例 3-15 所示。

例 3-15　`answerUnits` 的另一种写法

```
def pAnswerUnits: PartialFunction[Int, Int] =
    { case d: Int if d != 0 => 42 / d }

assert(pAnswerUnits(42) == 1)
//pAnswerUnits(0)
//scala.MatchError: 0 (of class java.lang.Integer)
```

例 3-15 联用 `case` 和防卫条件来共同限制参数的取值，并输出计算结果。两种写法有一处值得注意的区别，例 3-15 在除以 `0` 时得到的错误类型是 `MatchError`，不同于例 3-14 的 `ArithmeticException`，这是因为例 3-15 使用了模式匹配。

偏函数的使用范围不限于数值类型。我们可以把偏函数用在任何类型上，包括 `Any`。请看例 3-16 实现的一个递增函数。

例 3-16　用 Scala 语言定义一个递增函数

```
def inc: PartialFunction[Any, Int] =
    { case i: Int => i + 1 }

assert(inc(41) == 42)
//inc("Forty-one")
//scala.MatchError: Forty-one (of class java.lang.String)

assert(inc.isDefinedAt(41))
assert(! inc.isDefinedAt("Forty-one"))

assert(List(42) == (List(41, "cat") collect inc))
```

例 3-16 定义的偏函数接受任何类型的输入（`Any`），但只对其中特定的部分类型作出反应。例中我们对该偏函数调用了 `isDefinedAt()` 来判断它的取值范围，这是由于以 `case` 代码块方式实现的 `PartialFunction` trait 都隐含地定义了 `isDefinedAt()` 方法。我们在例 3-13 所见的 `map()` 和 `collect()` 的行为差异，可以从偏函数的行为得到解释：`collect()` 在设计的时候就考虑到传入偏函数的情况，会调用 `isDefinedAt()` 函数来鉴别集合元素是否符合取值条件，不符合的就被忽略掉了。

Scala 语言中的偏函数和部分施用函数英文原名比较接近，但以功能来说，它们根本就不在一个维度上。如果有需要的话，我们完全可以对一个偏函数进行部分施用。

3.3.5　一般用途

尽管有着微妙定义和繁琐的实现，但柯里化和部分施用都在现实的编程世界中拥有一席之地。

1. 函数工厂

我们在传统面向对象编程中会用到工厂方法的场合，正适合柯里化（以及部分施用）表现它的才干。我们可以用一个 Groovy 实现的简单加法函数来说明问题，请看例 3-17。

例 3-17　Groovy 实现的加法函数和递增函数

```
def adder = { x, y -> x + y}
def incrementer = adder.curry(1)

println "increment 7: ${incrementer(7)}" // 8
```

例中从 `adder()` 函数派生出了 `incrementer` 函数。

2. Template Method模式

GoF 模式集里面有一项 Template Method（模板方法）模式。其用意是在固定的算法框架内部安排一些抽象方法，为后续的具体实现保留一部分灵活性。部分施用和柯里化也可以起到相同的作用。部分施用技法注入当前已经确定的行为，留下未确定的参数给具体实现去发挥，其思路与模板方法这种面向对象的设计模式如出一辙。

本书第 6 章将会用一个例子来说明，若干设计模式（包括模板方法在内）怎样因为部分施用和其他函数式技法而失去了存在意义。

3. 隐含参数

当我们需要频繁调用一个函数，而每次的参数值都差不多的时候，可以运用柯里化来设置隐含参数。举个例子，我们在操作持久化框架的时候，每次都要在第一个参数里写上数据源的位置。而经过部分施用以后，我们就不需要反复地写出这个参数值了，如例 3-18 所示。

例 3-18　运用部分施用技法设置隐含参数值

```
(defn db-connect [data-source query params]
      ...)

(def dbc (partial db-connect "db/some-data-source"))

(dbc "select * from %1" "cust")
```

例 3-18 的 `dbc` 函数在操作数据的时候不需要再提供数据源，数据源已经自动设置好了。面向对象编程中"封装"概念的本质，也就是魔术般出现在每个函数里的隐含上下文 `this`，我们可以在函数式编程中加以模拟，用柯里化的方式把 `this` 传递给所有的函数，让 `this` 在使用者的面前隐藏起来。

3.4　递归

递归，按照维基百科的定义，是"以一种自相似的方式来重复事物的过程"。它也是我们

向运行时托付操作细节的一个例子，而且和函数式编程有着极为密切的联系。以具体的实践来说，递归是以一种带点计算机科学味道的方式来对一组事物进行迭代，让事物的集合反复对自身调用同样的方法，使集合随着每次迭代不断缩小，同时要始终小心地保证退出条件的有效性。很多时候，我们的问题核心就是对一个不断变短的列表反复地做同一件事，把递归用在这样的场合，写出来的代码就容易理解。

换个角度看列表

Groovy 大大加强了 Java 集合库的能力，其中就包括新增了许多函数式结构。Groovy 帮我们的第一个忙，是打开了看待列表的新角度，看起来微不足道的小事情却收到了意想不到的回报。

在 C 或类 C 语言（包括 Java）出身的开发者的头脑里面，列表概念通常会被塑造成一个带索引的集合。这个观察角度让我们很容易实现集合的迭代，甚至代码中不需要明确地用到索引，如例 3-19 的 Groovy 代码所示。

例 3-19　依靠（不一定直接露面的）索引来完成的列表遍历

```
def numbers = [6, 28, 4, 9, 12, 4, 8, 8, 11, 45, 99, 2]

def iterateList(listOfNums) {
  listOfNums.each { n ->
    println "${n}"
  }
}
println "迭代式的列表遍历"
iterateList(numbers)
```

Groovy 还提供了 `eachWithIndex()` 迭代子，要求传给它的代码块带有索引参数，适用于需要显式访问索引的场合。虽然例 3-19 中的 `iterateList()` 方法没有直接用到索引，但我的思维里面还是把集合想象成一排带编号的格子，就像图 3-1 的样子。

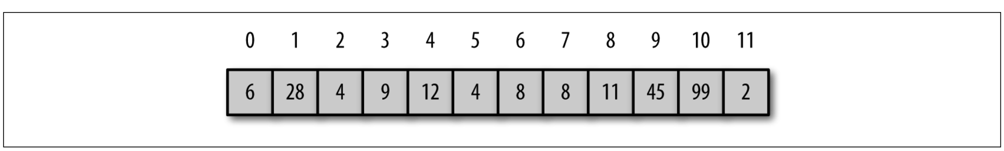


图 3-1：作为“带索引的格子”的列表形象

对于诸多函数式语言来说，它们眼中的列表形象有些不一样，所幸 Groovy 也持同样的观点。它们看到的不是带索引的格子，而是看成由列表的第一个元素（叫作头部）和列表的其余元素（叫作尾部）这两部分组合而成，如图 3-2 所示。

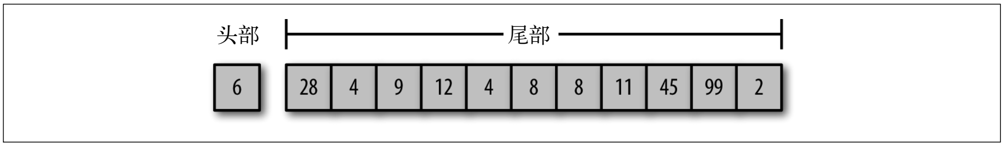


图 3-2：分成头和尾两部分的列表形象

把列表想象成头部和尾部的组合，有利于使用递归的方式来组织迭代，请看例 3-20。

例 3-20　以递归方式进行的列表遍历

```
def recurseList(listOfNums) {
  if (listOfNums.size == 0) return;
    println "${listOfNums.head()}"
    recurseList(listOfNums.tail())
}
println "\n递归式的列表遍历"
recurseList(numbers)
```

例 3-20 的 `recurseList()` 方法首先检查传入的列表里还有没有元素。如果没有，那就表示迭代工作已经完成，可以返回了。如果还有元素，那么用 Groovy 提供的 `head()` 方法取出第一个元素，把它打印出来，然后继续对列表的余下部分调用 `recurseList()` 方法。

递归操作往往受制平台而存在一些固有的技术限制，因此这种技法绝非万灵药。但对于长度不大的列表来说，递归操作是安全的。

我认为长远来看，还是应该更多地投入到良好的代码结构上，技术限制总会随着时间减少或者消失，就像我们在语言进化中看到的那样（详见第 5 章）。递归写法作为一种有缺点的代码结构，其优点并不那么直观。为了说清楚这一点，我们来看一个列表筛选的例子。例 3-21 是一个筛选方法，其参数除了一个列表，还有用来判断元素是否属于列表的一个谓词（即返回布尔值的一个测试）。

例 3-21　命令式写法的筛选函数，Groovy 实现

```
def filter(list, predicate) {
  def new_list = []
  list.each {
    if (predicate(it)) {
      new_list << it
    }
   }
   return new_list
}

modBy2 = { n -> n % 2 == 0}

l = filter(1..20, modBy2)
println l
```

例 3-21 完全是直截了当的写法：先创建一个新列表来存放希望保留的元素，然后对原列表

进行迭代，让谓词判定每个元素的去留，最后返回保留元素的列表。在调用 filter() 函数的时候，我们用了一个代码块来设置筛选条件。

如果用递归的方式来实现例 3-21 的筛选函数，将会是下面的样子，请看例 3-22。

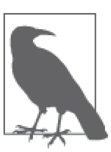
例 3-22　递归写法的筛选函数，Groovy 实现

```
def filterR(list, pred) {
  if (list.size() == 0) return list
  if (pred(list.head()))
    [] + list.head() + filterR(list.tail(), pred)
  else
    filterR(list.tail(), pred)
}

println "递归式的筛选"
println filterR(1..20, {it % 2 == 0})
//// [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

例 3-22 的 filter() 函数首先检查传入的列表的大小，若列表中已经没有元素，则返回列表。否则用筛选条件检查列表的头部，如果头部满足筛选条件，就把它放入列表（代码中用了一个空列表“[]”作为初始值来保证返回类型是正确的），不然就继续递归地对列表的尾部筛选下去。

例 3-21 与例 3-22 的区别凸显了一个重要的问题：谁来管理状态？在命令式的写法中，是“我”在管理状态。“我”必须创建一个叫 new_list 的新变量，“我”负责向新列表添加元素，“我”负责在筛选完成后返回新列表。而在递归写法中，是*语言*在管理返回值，它从递归栈里收集每次方法调用的返回结果，构造出最终的返回值。注意例 3-22 中 filter() 方法的每一条结束路径都返回到递归调用的上一层，随着栈中的调用一层一层地返回，各层得到的中间结果也自动汇集到一起。于是我们卸下了对 new_list 的管理责任，交由语言去替我们照料。



利用递归，把状态的管理责任推给运行时。

例 3-22 表现出来的筛选手法，如果用 Scala 那样的函数式语言来实现的话，会更加如鱼得水，请看例 3-23 结合了柯里化和递归的实现。

例 3-23　递归式的筛选函数，Scala 实现

```
object CurryTest extends App {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
```

```
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0) // ❶

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2)))  // ❷
  println(filter(nums, dividesBy(3)))
}
```

❶ 定义时已经指明函数将被柯里化使用。

❷ `filter` 要求传入一个集合（`nums`）和一个单参数的函数（柯里化之后 `dividesBy()` 函数就变成单参数了）。

Scala 的列表构造运算符“::”起到了提高代码可读性的作用，筛选通过和不通过这两种情形下返回结果的变动，表述得清晰易懂。例 3-23 是 Scala 文档用来说明递归和柯里化的例子。`filter()` 方法递归地使用参数 `p` 来筛选一个整数列表，其中参数 `p` 是一个布尔函数，或者按照函数式领域的术语叫作“谓词”（predicate）函数。`filter()` 方法检查列表是否为空，若是则直接返回；否则用谓词来检验列表的第一个元素（`xs.head`），判断是否应放入筛选后的列表。

如果头部满足谓词条件，那么就返回以该头部为首，再加上尾部的筛选结果组成的新列表。如果头部通不过谓词的检验，返回的就只有列表余下部分的筛选结果。

递归对开发者的解放效果或许不像垃圾收集那么显著，不过它切实地揭示了编程语言的一个重要的发展方向：通过移交“不确定因素”的控制权给运行时来消解它们。如果我们不准插手列表操作的中间结果，那么就不会引入那些在交互中产生的错误。

尾调用优化

递归没有成为一种平常的操作，其中一个主要原因是栈的增长。递归操作一般的实现方式，都是把中间结果放在栈里，于是没有为递归专门优化的语言就会遇到栈溢出的问题。而像 Scala、Clojure 这些语言则各自采用了不同的方式来规避这方面的局限。开发者也可以在这个问题上出一点力，使用尾调用优化（tail-call optimization）的写法来帮助运行时克服栈的增长问题。当递归调用是函数执行的最后一个调用的时候，运行时往往可以在栈里就地更新，而不需要增加新的栈空间。

很多函数式语言（如 Erlang，http://erlang.org/）实现了没有栈增长的尾递归。Erlang 用尾递归来实现长时间运行的进程，相当于运行在应用里面的一系列微服务，它们从别的进程接收消息，并按照消息中的要求来代表别的进程执行任务。这些接收消息并受消息左右的尾递归循环还有调整微服务内部状态的能力，因为对不可变的当前状态的任何作用结果，都可以放在表示新状态的变量里传入下一轮递归而生效。考虑到 Erlang 令人赞叹的容错能力，很可能有一些尾递归循环已经在生产系统中运行了数年而从未中断。

我敢说大多数读者在日常的编程中根本就不用递归，甚至连尝试的想法都没有。造成这样的局面，应该部分地归咎于大多数命令式语言呆滞的语法配合，让一件不太容易的事情变得难上加难。函数式语言的简洁语法和灵活配合，才使递归成为简单可行的代码重用选项之一。

3.5 Stream和作业顺序重排

从命令式风格转变为函数式风格还有一个潜在的好处，那就是运行时有能力在涉及效率的问题上替我们做决定。

我们可以把第 2 章用过的“公司业务过程”例子再拿出来看一遍，其 Java 8 实现如例 3-24 所示，其中只做了一点微小的改动。

例 3-24 公司业务过程的 Java 8 实现

```
public String cleanNames(List<String> names) {
    if (names == null) return "";
    return names
         .stream()
         .map(e -> capitalize(e))
         .filter(n -> n.length() > 1)
         .collect(Collectors.joining(","));
}
```

眼尖的读者会注意到我在这一版的 `cleanNames()` 里面调换了操作的顺序（与第 2 章的例 2-4 相比），`map()` 操作被提到了 `filter()` 的前面。按照命令式的思路，我们本能地就会把筛选操作放在映射操作的前面，这样 `map` 需要操作的列表会比较小，可以减少工作量。但是实际上很多函数式语言（包括 Java 8 乃至 Functional Java 框架）都提供了 `Stream` 抽象。`Stream` 很多方面的行为都与集合相似，但它不像集合那样事先就备妥所有的值，而是需要的时候才让数据从源头“流”向目的地。例 3-24 的数据源头是 `names` 集合，最终目的地（或者叫终结操作）是 `collect()`。处在中间的 `map()` 和 `filter()` 都是缓求值（lazy）的操作，它们会被尽量地推迟执行。实际上在下游的终结操作“发出要求”之前，它们都不会产生任何结果。

聪明的运行时会替我们重新安排缓求值操作的执行顺序。例 3-24 将在运行时的主持下调换其缓求值操作的顺序，让筛选操作先于映射操作执行，以取得最佳的运算效率。在使用 Java 平台上的各种函数式方案的时候，我们必须保证传给 `filter()` 等函数的 lambda 块不存在副作用，否则可能导致无法预料的结果。

允许运行时发挥其优化能力的做法，再次印证了我们关于交出控制权的观点：放弃对繁琐细节的掌控，关注问题域，而非关注问题域的实现。

我们会在第 4 章继续探讨缓求值的问题，Java 8 的 stream 特性则放在第 7 章讨论。

第 4 章

用巧不用蛮

我们转换范式的收获，表现在费更少的力气完成更多的事情。很多函数式编程构造的目的只有一个：从频繁出现的场景中消灭掉烦人的实现细节。

这一章，我们要讨论函数式语言的两种常见特性：记忆（memoization）和缓求值（laziness）。

4.1　记忆

“memoization”这个词是英国的人工智能研究者 Donald Michie 生造出来的，指的是在函数级别上对需要多次使用的值进行缓存的机制。目前来说，函数式编程语言普遍都支持记忆特性，有些是直接内建在语言里，也有一些需要开发者自行实现，但实现起来相对容易。

记忆可以用在这样的场合。假设我们有一个反复调用的函数，需要挖掘它的性能潜力。增加一个内部缓存是很容易想到的方案。每次我们根据一组特定参数求得结果之后，就用参数值做查找用的键，把结果缓存起来。以后当函数又遇到相同参数的时候，就不需要重新计算一遍了，可以直接返回缓存的结果。这种缓存函数计算结果的做法，是计算机科学里一种典型的折衷方案：用更多的内存（我们一般不缺内存）去换取长期来说更高的效率。

只有纯（pure）函数才可以适用缓存技术。纯函数是没有副作用的函数：它不引用其他值可变的类字段，除返回值之外不设置其他的变量，其结果完全由输入参数决定。`java.lang.Math` 类里面的方法都是纯函数的绝好例子。很显然，只有在函数对同样一组参数总是返回相同结果的前提下，我们才可以放心地使用缓存起来的结果。

4.1.1 缓存

缓存是很常见的一种需求（同时也是制造隐晦错误的源头）。在这一节里，我们首先分两种情况去剖析函数缓存的用法，一种是类内部缓存，另一种是外部调用。然后详细说明缓存的两种实现方式，一种是手工进行状态管理，另一种是采用记忆机制。

1. 方法级别的缓存

上一章我们用了完美数分类问题来充当考校不同方案的试验台。判定数字归属的工作由 `Classifier` 类负责，我们可以想见其中一种典型的用例，是让同一个数字把几个分类方法都跑一遍。就像下面的代码一样：

```
if (Classifier.isPerfect(n)) print "!"
else if (Classifier.isAbundant(n)) print "+"
else if (Classifier.isDeficient(n)) print "-"
```

按照先前的实现，被调用到的每一个分类方法，都只能够重新计算真约数和。这种情况恰好是*类内部缓存*的应用范本：在常规的使用中，每检查一个数字，都要调用 `sumOfFactors()` 方法若干次。原来的实现方案对于一个频繁出现的用例来说过于低效。

2. 缓存求和结果

再利用已有的工作成果是提高代码效率的办法之一。因为真约数和的计算成本高昂，所以我们希望每个数字只计算一次。照着这样的思路，我们建立了一个存放计算结果的缓存，如例 4-1 所示。

例 4-1　缓存求和结果

```
class ClassifierCachedSum {
  private sumCache = [:]

  def sumOfFactors(number) {
    if (! sumCache.containsKey(number)) {
      sumCache[number] = factorsOf(number).sum()
    }
   return sumCache[number]

  }
// 其余代码不变……
```

例 4-1 增加了一个和类一起初始化的散列 `sumCache`。在 `sumOfFactors()` 方法中，我们首先检查传入的参数是否已经在缓存里有对应的计算结果，有的话直接返回，否则才执行昂贵的计算，并在返回之前先把求和结果置入缓存。

这段代码比原来的复杂，但结果可以证明物有所值。只要把各个例子都按照例 4-2 的格式跑一遍测试就清楚了。

例 4-2　优化前的速度测试

```
def static final TEST_NUMBER_MAX = 5000

@Test
void mashup() {
  println "Test for range 1-${TEST_NUMBER_MAX}"
  print "未优化："
  start = System.currentTimeMillis()
  (1..TEST_NUMBER_MAX).each {n ->
    if (Classifier.isPerfect(n)) print '!'
    else if (Classifier.isAbundant(n)) print '+'
    else if (Classifier.isDeficient(n)) print '-'
  }
  println "\n\t ${System.currentTimeMillis() - start} ms"
  print "未优化(第二次运行)："
  start = System.currentTimeMillis()
  (1..TEST_NUMBER_MAX).each {n ->
    if (Classifier.isPerfect(n)) print '!'
    else if (Classifier.isAbundant(n)) print '+'
    else if (Classifier.isDeficient(n)) print '-'
}
println "\n\t ${System.currentTimeMillis() - start} ms"
```

例 4-2 的运行结果如表 4-1 所示，数据证明了缓存的效果。

表4-1：取值范围从1到1000的测试结果

版本	结果（数字越小越好）
未优化	577 ms
未优化（第二次运行）	280 ms
缓存求和结果	600 ms
缓存求和结果（第二次运行）	50 ms

按照表中的数据，未优化的版本首次运行耗时 577 毫秒，相比之下，缓存版本首次运行耗时 600 毫秒。两者的差别不明显，但可以看出建立缓存额外了耗费一点儿时间。在第二次运行的时候，未优化版本耗时减少到了 280 毫秒。两次运行的时间差异可以归结于垃圾收集等环境因素的影响。缓存版本的第二次运行表现出戏剧性的速度提升，仅耗时 50 毫秒。因为第二次运行的时候，所有计算结果都已经在缓存里了，所以数字所反映的只不过是我们读取散列的速度。第一次运行的时候，有无缓存的差别微不足道，而第二次运行的情况则大相径庭。这种情况是*外部缓存*的范本：调用方受益于缓存起来的计算结果，才有了第二次运行的高速。

缓存求和结果成效斐然，但也付出了一些代价。`ClassifierCachedSum` 不可以再纯粹由静态方法组成。类中的缓存就代表类有了状态，所有与缓存打交道的方法都不可以是静态的，于是产生了更多的连锁效应。我们可以安排 `Singleton` 模式来解决一部分影响，但这样做本身就提高了复杂性，还会带来一箩筐的测试问题。由于是我们自己来操控缓存，那就有责任保障其正确性（比如做一些单元测试）。缓存可以提高性能，但缓存有代价：它提高

了代码的非本质复杂性和维护负担。

3. 缓存一切结果

既然缓存求和结果大大提高了代码的性能，何不试试把所有可能出现的中间结果都缓存起来呢？例 4-3 实践了这个想法。

例 4-3　缓存所有的计算结果

```
class ClassifierCached {
  private sumCache = [:], factorCache = [:]

  def sumOfFactors(number) {
    if (! sumCache.containsKey(number))
      sumCache[number] = factorsOf(number).sum()
    sumCache[number]
  }

  def isFactor(number, potential) {
    number % potential == 0;
  }

  def factorsOf(number) {
    if (! factorCache.containsKey(number))
      factorCache[number] = (1..number).findAll {isFactor(number, it)}
    factorCache[number]
  }

  def isPerfect(number) {
    sumOfFactors(number) == 2 * number
  }

  def isAbundant(number) {
    sumOfFactors(number) > 2 * number
  }

  def isDeficient(number) {
    sumOfFactors(number) < 2 * number
  }

}
```

例 4-3 的 ClassifierCached 类除了缓存真约数和的计算结果，还缓存了每个数的约数。性能优化的测试成绩如表 4-2 所示。

表4-2：取值范围从1到1000的测试结果

版本	结果（数字越小越好）
未优化	577 ms
未优化（第二次运行）	280 ms
缓存求和结果	600 ms
缓存求和结果（第二次运行）	50 ms
全部缓存	411 ms
全部缓存（第二次运行）	38 ms

缓存全部结果的版本（作为与前面的测试对象完全不同的新类和新实例变量）首次运行耗时 411 毫秒，缓存填充完毕后的第二次运行更达到了惊人的 38 毫秒。尽管成绩优秀，但这种写法应付不了大规模的数据。当我们把测试的取值范围增加到 8000 个数字，马上就变成了下面的糟糕结果：

```
java.lang.OutOfMemoryError: Java heap space
        at java.util.ArrayList.<init>(ArrayList.java:112)
……更多不想见到的坏消息……
```

这几次测试告诉我们，负责编写缓存代码的开发者不仅要顾及代码的正确性，连它的执行环境也要考虑在内。所谓“不确定因素”说的就是这样的东西：代码中的状态，开发者不仅要费心照应它，还要条分缕析它的一切明暗牵连。好在很多语言已经有了突破困境的办法，例如记忆机制。

4.1.2　引入“记忆”

函数式编程费了很大的力气来遏制不确定因素，并为此在运行时里内建了多种重用机制。“记忆”是其中的一种特性，它作为编程语言的固有设施，自动地缓存重复出现的函数返回值。换句话说，记忆特性会自动地提供我们写在例 4-1 和例 4-3 里的那些的代码。很多现代语言都支持记忆特性，其中就有 Groovy。

Groovy 语言记忆一个函数的办法是，先将要记忆的函数定义成闭包，然后对该闭包执行 `memoize()` 方法来获得一个新函数，以后我们调用这个新函数的时候，其结果就会被缓存起来。

“记忆一个函数”这件事情，运用了所谓的“元函数”技法：我们操纵的对象是函数本身，而非函数的结果。第 3 章讨论的柯里化也属于一种元函数技法。Groovy 把记忆特性内建在它的 `Closure` 类里面，其他语言各有各的实现方式。

为了让 `sumOfFactors()` 得到像例 4-1 那样的缓存能力，我们记忆了 `sumOfFactors()` 方法，请看例 4-4。

例 4-4　记忆求和结果

```
package com.nealford.ft.memoization

class ClassifierMemoizedSum {
  def static isFactor(number, potential) {
    number % potential == 0;
  }

  def static factorsOf(number) {
    (1..number).findAll { i -> isFactor(number, i) }
  }
```

```
  def static sumFactors = { number ->
    factorsOf(number).inject(0, {i, j -> i + j})
  }
  def static sumOfFactors = sumFactors.memoize()

  def static isPerfect(number) {
    sumOfFactors(number) == 2 * number
  }

  def static isAbundant(number) {
    sumOfFactors(number) > 2 * number
  }

  def static isDeficient(number) {
    sumOfFactors(number) < 2 * number
  }
}
```

例 4-4 按照代码块的格式（注意看 = 和参数的写法）来实现 `sumFactors()` 方法。方法本身平平无奇，说不定可以直接在哪个库里找到现成的。为了记忆 `sumFactors()`，我们对它调用了 `memoize()` 方法，并将返回的新函数命名为 `sumOfFactors`。

对这个记忆了部分函数的版本进行测试，得到如表 4-3 所示的数据。

表4-3：取值范围从1到1000的测试结果

版本	结果（数字越小越好）
未优化	577 ms
未优化（第二次运行）	280 ms
缓存求和结果	600 ms
缓存求和结果（第二次运行）	50 ms
全部缓存	411 ms
全部缓存（第二次运行）	38 ms
部分记忆	228 ms
部分记忆（第二次运行）	60 ms

部分记忆版本同样在第二次运行中得到了极大的速度提升，丝毫不逊色于手工编写的缓存效果，而我们所付出的劳动，仅仅是修改了两行代码（把 `sumFactors()` 的定义从函数改成代码块，以及增加了指向代码块被记忆实例的引用 `sumOfFactors()`）。

在手工实现的版本里，我们尝试过缓存所有可能被重用的计算结果。作为对比，我们也用记忆的方式来实现一次。例 4-5 是记忆了全部计算结果的版本，其测试数据如表 4-4 所示。

例 4-5　记忆所有计算结果

```
package com.nealford.ft.memoization

class ClassifierMemoized {
```

```
  def static dividesBy = { number, potential ->
    number % potential == 0
  }
  def static isFactor = dividesBy.memoize()

  def static factorsOf(number) {
    (1..number).findAll { i -> isFactor.call(number, i) }
  }

  def static sumFactors = { number ->
    factorsOf(number).inject(0, {i, j -> i + j})
  }
  def static sumOfFactors = sumFactors.memoize()

  def static isPerfect(number) {
    sumOfFactors(number) == 2 * number
  }

  def static isAbundant(number) {
    sumOfFactors(number) > 2 * number
  }

  def static isDeficient(number) {
    sumOfFactors(number) < 2 * number
  }
}
```

表4-4：取值范围从1到1000的测试结果

版本	结果（数字越小越好）
未优化	577 ms
未优化（第二次运行）	280 ms
缓存求和结果	600 ms
缓存求和结果（第二次运行）	50 ms
全部缓存	411 ms
全部缓存（第二次运行）	38 ms
部分记忆	228 ms
部分记忆（第二次运行）	60 ms
全部记忆	956 ms
全部记忆（第二次运行）	19 ms

扩大记忆范围拖慢了第一次运行的速度，但后续运行的速度是所有版本中最快的——但这只是数据量很小的情况。随着数据量变大，它的性能也像例 4-3 的命令式缓存版本那样急剧下滑。事实上当数据量达到 8000 的时候，就出现了内存不足。命令式的版本要想防范这种陷阱，需要小心地查看警戒条件，注意执行环境是否超出安全范围——命令式编程的不确定因素又一次出现在我们面前。相比之下，通过记忆方式实现的例 4-5 修正起来十分简单，只需要在函数的层次上做改动。修改后的记忆版本可以轻松应付 10 000 条的数据量，测试结果见表 4-5。

表4-5：取值范围从1到10 000的测试结果

版本	结果（数字越小越好）
未优化	41 909 ms
未优化（第二次执行）	22 398 ms
记忆最多 1000 个结果	55 685 ms
记忆最多 1000 个结果（第二次运行）	98 ms

我们只需要用 `memoizeAtMost(1000)` 方法代替原来的 `memoize()`，就取得了表 4-5 的成绩。Groovy 和其他支持记忆特性的语言一样，提供了适合不同情况使用的多个记忆方法，如表 4-6 所示。

表4-6：Groovy语言提供的几个记忆方法

方法	说明
`memoize()`	将闭包转化为带缓存的实例
`memoizeAtMost()`	将闭包转化为带缓存的实例，且规定了缓存的数量上限
`memoizeAtLeast()`	将闭包转化为带缓存的实例，缓存大小可自动调整，且规定了缓存的数量下限
`memoizeBetween()`	将闭包转化为带缓存的实例，缓存大小可自动调整，且规定了缓存的数量上限和下限

在命令式的思路下，开发者是代码的主人（以及一切责任的承担者）。而函数式语言的思路是，为了操纵一些标准的构造，我们来制作一些通用的机件，有时候还在机件上设置若干调节旋钮（也就是函数的不同变体和参数的不同组合）。函数是语言的基本元素，因此函数层面上的优化会附带产生功能的提升。以我们举的几个例子来说，使用记忆特性的版本轻而易举地跑赢了手工编写缓存的版本。实际上，我们写出来的缓存绝不可能比语言设计者产生的更高效，因为语言设计者可以无视他们给语言定的规矩：开发者无法触碰的底层设施，不过是语言设计者手中的玩物，他们拥有的优化手段和空间是“凡人”无法企及的。但我们将缓存等问题交托给语言，不仅仅因为它的效率更高，更因为我们从此可以站在更高的抽象层次上去思考问题。

> 语言设计者实现出来的机制总是比开发者自己做的效率更高，因为他们可以不受语言本身的限制。

手工建立缓存的工作不算复杂，但它给代码增加了状态的影响和额外的复杂性。而借助函数式语言的特性，例如记忆，我们可以在函数的级别上完成缓存工作，只需要微不足道的改动，就能取得比命令式做法更好的效果。在函数式编程消除了不确定因素之后，我们得以专注解决真正的问题。

当然，我们不需要先写好类，再往上添加记忆层。`memoize()` 和它的兄弟们都是定义在 `Closure` 类里面的库函数。例 4-6 演示了直接内联声明函数记忆能力的用法。

例 4-6　内联声明函数的记忆能力，Groovy 实现

```
package com.nealford.javanext.memoizehashing

class NameHash {
  def static hash = {name ->
    name.collect{rot13(it)}.join()
  }.memoize()

  public static char rot13(s) {
    char c = s
    switch (c) {
      case 'A'..'M':
      case 'a'..'m': return c + 13
      case 'N'..'Z':
      case 'n'..'z': return c - 13
      default: return c
    }
  }

}
```

例 4-6 的 `rot13()` 替换加密算法（凯撒密码的一种）其实没有多大的计算量，只是为了演示，我们就假装值得为它增加缓存吧。注意例中将代码块赋值给 `hash` 变量的地方，函数定义语法在这里发生了小小的变化。在定义末尾，我们直接调用了 `memoize()` 方法，也就是说，这一次的函数没有一个不带记忆能力的对应版本。

我们来对这个被记忆的函数做一个单元测试，见例 4-7。

例 4-7　测试被记忆的散列函数

```
class NameHashTest extends GroovyTestCase {
  void testHash() {
    assertEquals("ubzre", NameHash.hash.call("homer")) }
}
```

例 4-7 特意写出了代码块的 `call()` 调用，这是必须的。一般来说，Groovy 提供的语法糖衣允许我们直接在变量名后加上一对圆括号来执行代码块里的内容，例如写成 `(NameHash.hash("Homer"))`，语言会帮我们在暗地里调用 `call()` 方法。但由于 Groovy 语言目前实现上的原因，像例中这种情况，必须通过显式的 `call()` 语法来调用被记忆的函数。

大多数函数式语言要么直接提供了记忆特性，要么实现起来极其轻松。例如 Clojure 语言就内建了记忆特性，我们可以通过语言内建的 `(memoize )` 函数为任意函数增加记忆能力。例如想记忆现有的一个 `(hash )` 函数，我们只要写成 `(memoize (hash "homer"))`，就可以得到该函数的缓存版本。例 4-8 用 Clojure 语言重新实现了一遍例 4-6 的散列算法。

例 4-8　Clojure 语言的记忆机制

```
(ns name-hash.core)
(use '[clojure.string :only (join split)])


(let [alpha (into #{} (concat (map char (range (int \a) (inc (int \z))))
                              (map char (range (int \A) (inc (int \Z))))))
      rot13-map (zipmap alpha (take 52 (drop 26 (cycle alpha))))]

  (defn rot13
    "对给定的输入字符串,求出经rot 13变换后的对应字符串。
     \"hello\" -> \"uryyb\""
    [s]
    (apply str (map #(get rot13-map % %) s))))

(defn name-hash [name]
  (apply str (map #(rot13 %) (split name #"\d"))))

(def name-hash-m (memoize name-hash))
```

例 4-7 在调用被记忆函数的时候，必须通过 `call()` 方法。而我们从上面的 Clojure 代码看到，被记忆方法的调用形式与一般的函数没有任何外在区别，因此对于方法的使用者来说，缓存以及背后的间接层次都是不可见的。

Scala 语言没有直接提供记忆机制，但它为集合提供的 `getOrElseUpdate()` 方法已经替我们承担了大部分的实现工作，请看例 4-9。

例 4-9　在 Scala 语言中实现记忆机制

```
def memoize[A, B](f: A => B) = new (A => B) {
    val cache = scala.collection.mutable.Map[A, B]()
    def apply(x: A): B = cache.getOrElseUpdate(x, f(x))
  }

def nameHash = memoize(hash)
```

例 4-9 用到的 `getOrElseUpdate()` 函数完美地符合我们构造缓存的需要：它要么返回符合条件的元素项，要么当不存在这样的元素时生成一个新的项。

被记忆的内容应该是值不可变的，这一点非常重要，有必要多说几遍。如果被记忆的函数在求解其返回结果的时候，需要依赖参数以外的任何因素，那么它的输出是不可预料的。如果被记忆的函数有副作用，那么当它直接返回缓存结果的时候，就跳过了产生副作用的那部分代码，没有执行。

> 请保证所有被记忆的函数：
> - 没有副作用
> - 不依赖任何外部信息

随着运行时的功能越来越完善，我们可以支配的运算能力也越来越强，像记忆这样的高级特性已经成为所有主流语言的日常配备。例如 Java 8 虽然没有直接提供记忆特性，但只要借助它新增的 lambda 特性，就可以轻松地实现记忆功能。

> 就算你对 Scala、Clojure 这些函数式语言毫无兴趣，宁可固守当前使用的语言，函数式编程还是会随着语言的演变而进入你的生活。

4.2 缓求值

缓求值（lazy evaluation）是函数式编程语言常见的一种特性，指尽可能地推迟求解表达式。缓求值的集合不会预先算好所有的元素，而是在用到的时候才落实下来，这样做有几个好处。第一，昂贵的运算只有到了绝对必要的时候才执行。第二，我们可以建立无限大的集合，只要一直接到请求，就一直送出元素。第三，按缓求值的方式来使用映射、筛选等函数式概念，可以产生更高效的代码。Java 8 以前的 Java 语言本身不支持缓求值，但平台上有一些框架和后继语言提供了这样的支持。

我们用例 4-10 的伪代码来做一些分析，它的工作是打印出列表的长度。

例 4-10　演示“非严格求值”的伪代码

```
print length([2+1, 3*2, 1/0, 5-4])
```

这段代码会得到怎样的执行结果，取决于所用编程语言的一项性质：它是*严格求值*（strict）的，还是*非严格求值*（non-strict）的（也叫缓求值，lazy）。在严格求值的编程语言里，执行（甚至编译）这段代码，会因为列表中的第三个元素而发生“被零除”异常。而在非严格求值的语言里，它会得出 4 的结果，准确地报告列表元素的数目。毕竟我们调用的方法叫作 `length()`，而不叫 `lengthAndThrowExceptionWhenDivByZero()`！常用的非严格求值语言有 Haskell。Java 虽然不属于这个阵营，但缓求值的思路仍然可以给我们带来好处，另外 Java 平台上的下一代语言，有一部分已经具备了更明显的缓求值特征。

4.2.1 Java语言下的缓求值迭代子

我们在 Java 语言下实现缓求值的概念，首先需要找到一个合适的数据结构作为支撑。我们的探索可以从例 4-11 的*素数*（只能被 1 和它本身整除的自然数）类开始。

例 4-11　寻找素数，Java 实现

```
package com.nealford.functionalthinking.primes;

import java.util.HashSet;
```

```
import java.util.Set;

import static java.lang.Math.sqrt;

public class Prime {

    public static boolean isFactor(final int potential, final int number) {
        return number % potential == 0;
    }

    public static Set<Integer> getFactors(final int number) {
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < sqrt(number) + 1; i++)
            if (isFactor(i, number)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    public static int sumFactors(final int number) {
        int sum = 0;
        for (int i : getFactors(number))
            sum += i;
        return sum;
    }

    public static boolean isPrime(final int number) {
        return sumFactors(number) == number + 1;
    }

    public static Integer nextPrimeFrom(final int lastPrime) {
        int candidate = lastPrime + 1;
        while (!isPrime(candidate)) candidate++;
        return candidate;
    }

}
```

Java 语言本身不提供缓求值的集合，但这并不妨碍我们实现一个特殊的 `Iterator` 来模拟缓求值集合。有了例 4-11 的准备工作，我们继续构造一个能够按需要返回下一个素数的迭代子，如例 4-12 所示。

例 4-12　素数迭代子，Java 实现

```
package com.nealford.ft.laziness;

import java.util.Iterator;

public class PrimeIterator implements Iterator<Integer> {
    private int lastPrime = 1;
```

```
        @Override
        public boolean hasNext() {
            return true;
        }

        @Override
        public Integer next() {
            return lastPrime = Prime.nextPrimeFrom(lastPrime);
        }

        @Override
        public void remove() {
            throw new RuntimeException("颠覆宇宙真理的异常！");
        }
    }
```

一般来说，开发者会把迭代子想象成在背后有一个存储数据的集合，实际上任何对象只要支持了 `Iterator` 接口，就可以算是一个迭代子。例 4-12 里面的 `hasNext()` 方法总是返回 `true`，这是因为数学上可以证明，素数有无穷多个。`remove()` 方法在这里没有意义，因此我们让它在意外被调用的时候抛出异常。承担主要工作的 `next()` 方法在它仅有一行的方法体里面做了两件事情。首先，它通过调用我们在例 4-11 中实现的 `nextPrimeFrom()` 方法，根据上一个素数来找到下一个素数。其次，它利用 Java 用一条语句来同时完成赋值和返回操作的能力，更新了内部的 `lastPrime` 字段。

4.2.2 使用Totally Lazy框架的完美数分类实现

有人可能会想，除非遥遥无期地等待公司升级到 Java 8，不然 Java 语言是注定与简洁的函数式代码无缘的。我们确实不可能给旧版本的 Java 安上真正的高阶函数，不过有些框架创造性地运用泛型、匿名类和静态导入（static import）机制，也能部分地展现出函数式编程的优点。

我们在第 2 章讨论过*完美数分类*的例子，例 4-13 使用 Totally Lazy 框架重新实现了一遍。Totally Lazy（https://code.google.com/p/totallylazy）是一套 Java 框架，它以一种稍显啰嗦的方式，在 Java 语言下实现了函数式风格的语法。

例 4-13　使用 Totally Lazy Java 框架实现的完美数分类

```
import com.googlecode.totallylazy.Predicate;
import com.googlecode.totallylazy.Sequence;

import static com.googlecode.totallylazy.Predicates.is;
import static com.googlecode.totallylazy.numbers.Numbers.*;
import static com.googlecode.totallylazy.predicates.WherePredicate.where;

public class NumberClassifier {
    public static Predicate<Number> isFactor(Number n) {
        return where(remainder(n), is(zero));            ❶
    }
```

```
public static Sequence<Number> getFactors(final Number n) {
    return range(1, n).filter(isFactor(n));
}

public static Sequence<Number> factors(final Number n) {
    return getFactors(n).memorise();
}

public static Number aliquotSum(Number n) {
    return subtract(factors(n).reduce(sum), n);
}

public static boolean isPerfect(Number n) {
    return equalTo(n, aliquotSum(n));
}

public static boolean isAbundant(Number n) {
    return greaterThan(aliquotSum(n), n);
}

public static boolean isDeficient(Number n) {
    return lessThan(aliquotSum(n), n);
}}
```

❶ `remainder` 等函数和 `where` 等谓词都是框架提供的。

开头的连串静态导入让我们得以省略频繁出现的类前缀，减少了行文中的干扰。经过这样的处理，代码已经不像典型的 Java 语句，但可读性很高。Totally Lazy 框架对 Java 的补益不可能越出 Java 语法的界限，因此它没有运用运算符重载，而通过增加适当的方法来改善表现力。于是 `num % i == 0` 就写成了 `where(remainder(n), is(zero))` 的形式。

Totally Lazy 的简便语法有一部分是受到 JUnit 测试框架（http://junit.org/）的扩展库 Hamcrest（https://code.google.com/p/hamcrest）的启发，甚至直接使用了 Hamcrest 的一些类。在 Totally Lazy 的 `remainder()` 方法和 Hamcrest 的 `is()` 方法携手之下，我们用一次 `where()` 调用来完成了 `isFactor()` 方法的工作。`factors()` 方法也类似地变成对 `range()` 对象进行的一次 `filter()` 调用。约数的求和工作则由我们现在已经很熟悉的 `reduce()` 方法来完成。由于 Java 不支持运算符重载，求解 `aliquotSum` 所需的减法运算也只好变成了对 `subtract()` 方法的调用。最后，`isPerfect()` 方法使用 Hamcrest 提供的 `equalTo()` 方法来判断目标数本身是否等于它的真约数和。

Totally Lazy 利用 Java 语言中不甚起眼的静态导入特性，出色地营造了富于可读性的代码。很多开发者武断地相信 Java 是一种糟糕的内部 DSL（领域专用语言）宿主，Totally Lazy 戳破了这种论调。缓求值也是 Totally Lazy 积极运用的原则之一，一切操作都被尽可能地推迟。

我们如果希望建立更传统一些的缓求值数据结构，高阶函数会是很重要的一件工具。

4.2.3 Groovy语言的缓求值列表

缓求值列表是函数式语言普遍具备的特性，这种列表只在需要的时刻才产生其中的内容。缓求值列表的作用之一是暂缓初始化昂贵的资源，除非到了绝对必要的时候。缓求值列表还可以用来构建无限序列，也就是没有上边界的列表。如果需求上没有预先限定列表的大小，那么我们可以让它根据需要来变化。

我们可以先用一个例子来体会一下 Groovy 中缓求值列表的用法，然后再考虑怎样实现一个缓求值列表。请看例 4-14。

例 4-14　缓求值列表在 Groovy 中的应用

```
def prepend(val, closure) { new LazyList(val, closure) }

def integers(n) { prepend(n, { integers(n + 1) }) }

@Test
public void lazy_list_acts_like_a_list() {
    def naturalNumbers = integers(1)
    assertEquals('1 2 3 4 5 6 7 8 9 10', naturalNumbers.getHead(10).join(' '))
    def evenNumbers = naturalNumbers.filter { it % 2 == 0 }
    assertEquals('2 4 6 8 10 12 14 16 18 20', evenNumbers.getHead(10).join(' '))
}
```

例 4-14 的第一个方法 `prepend()` 创建了一个缓求值列表，供后续代码向列表追加数据。熟悉函数式语言的读者可能会注意到，这个方法就相当于函数式语言中一般叫作 `cons()` 的列表构建函数。下一个方法 `integers()` 利用 `prepend()` 方法来产生并返回一个整数列表。我们传给 `prepend()` 的两个参数是列表的初始元素值和用来产生下一个元素的代码块。`integers()` 就像一个制造整数列表的工厂，初始值排在流水线的打头位置上，后续的值从制造机构里源源生产出来。

从列表中取出值要通过 `getHead()` 方法，它按照参数里指定的数目，取出列表头部的若干元素。例 4-14 中的 `naturalNumbers` 是表示所有自然数的缓求值列表，我们调用 `getHead()` 方法来取得它的一个子集，并在参数里指定需要的数目。我们得到的返回值正是断言中列出的从 1 到 10 的自然数。接下来的用例是通过 `filter()` 来获得一个偶数的缓求值列表，然后用 `getHead()` 来取得前 10 个偶数。

例 4-15 给出了 `LazyList` 的实现。

例 4-15　`LazyList` 的 Groovy 实现

```
package com.nealford.ft.allaboutlists

class LazyList {
    private head, tail

    LazyList(head, tail) {
```

```
        this.head = head;
        this.tail = tail
    }

    def LazyList getTail() { tail ? tail() : null }

    def List getHead(n) {
        def harvestedValues = [];
        def current = this
        n.times {
            harvestedValues << current.head
            current = current.tail
        }
        harvestedValues
    }

    def LazyList filter(Closure p) {
        if (p(head))
            p.owner.prepend(head, { getTail().filter(p) })
        else
            getTail().filter(p)
    }
}
```

缓求值列表由头部 head 和尾部 tail 两部分构成，构造函数明显地体现了这一点。getTail() 方法检查尾部，若不是 null 则执行它。getHead() 方法负责收集要返回的元素，它每次从列表头部摘下一个现成的元素，同时令尾部生产一个新的元素，并按照需要返回的元素数目，调用 n.times {...} 来重复上述过程，最后返回收集好的所有值。

缓求值列表特别适用于资源的生产成本较高的情况，例如创建一个完美数的列表。

完美数的缓求值列表

我们继续用一个熟悉的例子——第 2 章的完美数分类问题——来做试验对象。迄今为止我们尝试过的所有实现方式都有一个共同缺点，那就是必须先给定要分类的数字。而这一次，我们希望得到一个完美数的缓求值列表。例 4-16 给出了一个富于函数式风格，且十分紧凑的实现。

例 4-16　删繁就简的完美数分类程序，Groovy 实现

```
package com.nealford.ft.allaboutlists

import static com.nealford.ft.allaboutlists.NumberClassification.*

def enum NumberClassification {
  PERFECT, ABUNDANT, DEFICIENT
}

class NumberClassifier {
  static def factorsOf(number) {
    (1..number).findAll { i -> number % i == 0 }
  }
```

```
  static def classify(number) {
    switch (factorsOf(number).inject(0, { i, j -> i + j })) {
      case { it < 2 * number }: return DEFICIENT
      case { it > 2 * number }: return ABUNDANT
      case { it == 2 * number }: return PERFECT
    }
  }

  static def isPerfect(number) {
    classify(number) == PERFECT
  }

  static def nextPerfectNumberAfter(n) {
    while (!isPerfect(++n));
    n
  }
}
```

例 4-16 构造了一个紧凑的 `classify()` 方法，它在隐含变量 `it` 上逐一验证各条分类规则，然后返回一个 `NumberClassification` 枚举来表明检查的结果。新出现的方法 `nextPerfectNumber()` 在内部使用 `isPerfect()` 来寻找大于参数值 `n` 的下一个完美数。就算 `n` 的数字不大，这个方法也需要执行很长时间才能得出结果（尤其是提供的代码未经任何优化），况且完美数本就分布得比较稀疏。

有了这个新版本的 `NumberClassifier`，我们就可以建立完美数的缓求值列表了，如例 4-17 所示。

例 4-17　推迟初始化的完美数列表

```
def perfectNumbers(n) { prepend(n,
   { perfectNumbers(nextPerfectNumberAfter(n)) }) };

 @Test
public void infinite_perfect_number_sequence() {
    def perfectNumbers = perfectNumbers(nextPerfectNumberAfter(1))
    assertEquals([6, 28, 496], perfectNumbers.getHead(3))
}
```

通过例 4-15 定义的 `prepend()` 方法，我们构造了一个完美数的列表，其中头部是列表的初始值，尾部则是能够计算出下一个完美数的的代码块。列表首先经过初始化，求出大于 1 的第一个完美数（经过静态导入，我们可以使用更简短的写法来调用 `NumberClassifier.nextPerfectNumberFrom()`）。最后我们让初始化好的列表返回了前三个完美数。

查找完美数的计算成本十分高昂，我们自然会希望尽量少做这样的计算。在缓求值列表的帮助下，我们能够把计算推迟到最适当的时刻才去执行。

当“列表”被抽象成“带编号的格子”的时候，我们不容易设想无限序列应该是什么样子。而由“第一个元素”和“剩下的部分”构成的列表形象，则会更多地促使我们越过列

表的结构去考虑其中的元素，进而有机会萌发出像缓求值列表这一类的新思路。

4.2.4 构造缓求值列表

上文提过，编程语言可以分成急切求解所有表达式的严格求值，以及推迟求解直到最后关头的缓求值两个类别。Groovy 本质上是一种严格求值的语言，但如果我们用闭包递归地将一个严格求值的列表层层包裹起来，就可以将之变换成缓求值列表。我们只要推迟执行闭包，也就推迟了对后续元素值的求解。

Groovy 把严格求值的空列表表示成一个数组，写成一对没有内容的方括号：`[]`。如果用闭包把它括起来，就变成了一个缓求值的空列表：

```
{-> [] }
```

如果我们需要向列表添加一个元素，可以添加在列表的前端，形成新的列表，然后再次让它变成缓求值的：

```
{-> [ a, {-> [] } ] }
```

向列表前端添加元素的方法，传统上一般命名为 `prepend` 或者 `cons`。我们继续重复这个添加新元素的过程，添加三个元素（`a`、`b`、`c`）之后，得到如下结果：

```
{-> [a, {-> [b, {-> [ c, {-> [] } ] } ] } ] }
```

这里的语法相当笨拙，但只要我们掌握了其中的原理，就可以在 Groovy 语言下构造一个完整实现了传统接口的缓求值集合，如例 4-18 所示。

例 4-18 在 Groovy 语言中构造一个缓求值列表

```
class PLazyList {
  private Closure list

  private PLazyList(list) {
    this.list = list
  }

  static PLazyList nil() {
    new PLazyList({-> []})
  }

  PLazyList cons(head) {
    new PLazyList({-> [head, list]})
  }

  def head() {
    def lst = list.call()
    lst ? lst[0] : null
  }
```

```
  def tail() {
    def lst = list.call()
    lst ? new PLazyList(lst.tail()[0]) : nil()
  }

  boolean isEmpty() {
    list.call() == []
  }

  def fold(n, acc, f) {
    n == 0 || isEmpty() ? acc : tail().fold(n - 1, f.call(acc, head()), f)
  }

  def foldAll(acc, f) {
    isEmpty() ? acc : tail().foldAll(f.call(acc, head()), f)
  }

  def take(n) {
    fold(n, []) {acc, item -> acc << item}
  }

  def takeAll() {
    foldAll([]) {acc, item -> acc << item}
  }

  def toList() {
    takeAll()
  }
}
```

例 4-18 设置了一个私有的构造函数，`nil()` 方法调用该构造函数来生成一个空列表，且在调用时传入了一个空列表作为起始。`cons()` 方法将其参数作为新元素添加到列表的头部，然后把添加结果装进一个闭包。

接下来的三个方法是实现列表遍历的必需品。`head()` 方法返回列表的第一个元素，`tail()` 返回余下的部分，即由除第一个元素以外的所有元素构成的子列表。在这两个方法里面，我们都对闭包执行了 `call()`，这是为了迫使闭包里的缓求值操作进入执行。当我们开始向列表索取元素的时候，它就不能继续推迟操作了，必须立即开始执行来响应要求。`isEmpty()` 方法顾名思义，是用来检查列表中是否还有待求解的元素项。

其余的方法都是一些操纵列表的高阶函数。`fold()` 和 `foldAll()` 方法负责执行我们熟悉的折叠操作。前面的内容已经展示过它们的用法，不过例 4-18 是我们首次完全用闭包来构造这样一个递归的函数定义。`foldAll()` 方法检查列表是否为空，若为空则返回 `acc`（代表累积量 accumulator，即折叠操作的初始值）。如果列表不为空，那么它在列表尾部 `tail()` 上递归地调用 `foldAll()`，并经参数传入累积量和列表头部。`f` 参数所代表的函数规定要有两个参数，且返回单一值；在我们将一个元素“折叠”到它的相邻元素上的时候，具体执行哪些动作就由这个函数来定义。

例 4-19 演示了怎样构建和操纵我们设计的这个列表。

例 4-19　缓求值列表用法演示

```
def lazylist = PLazyList.nil().cons(4).cons(3).cons(2).cons(1)
println(lazylist.takeAll()) // [1, 2, 3, 4]
println(lazylist.foldAll(0, {i, j -> i + j})) // 10
lazylist = PLazyList.nil().cons(1).cons(2).cons(4).cons(8)
println(lazylist.take(2)) // [8, 4]
```

例 4-19 首先演示列表的创建，做法是在空列表上连续地调用 `cons()` 来添加新的值。接着用 `takeAll()` 取出了全部元素，输出结果有一点值得注意，各元素的出现次序与它们被放进列表的次序是相反的。请记住 `cons()` 实际上是一个向前追加（prepend）的操作，新元素被放置在列表的最前面。我们传给 `foldAll()` 方法一个代码块 `{i, j → i + j}`，当 `foldAll()` 执行这个代码块的时候，就相当于对列表进行了求和运算。最后我们调用 `take()` 方法来迫使列表立即求解它的前两个元素。

现实中不会直接按照例子里的方式去实现缓求值列表，通常会避免使用递归，而且会包含更丰富、灵活的列表操纵方法。但至少我们从概念上知道了实现背后的原理，对学习和使用都是有利的。

4.2.5　缓求值的好处

缓求值列表有几个好处。第一，我们可以用它创建无限长度的序列。由于不需要求解还没用到的元素值，我们可以用缓求值集合来建模无限列表，例 4-16 已经演示过这样的用法。

第二个优点是减少占用的存储空间。假如能够用推导的方法得到后续的值，那就不必预先存储完整的列表了——这是牺牲速度来换取存储空间的做法。是否采用缓求值集合，取决于我们如何权衡元素值的存储和计算的花费。

第三点是缓求值集合的关键优势所在，缓求值集合有利于运行时产生更高效率的代码。请看例 4-20。

例 4-20　查找“回文词”，Groovy 实现

```
def isPalindrome(s) {
  def sl = s.toLowerCase()
  sl == sl.reverse()
}

def findFirstPalindrome(s) {
  s.tokenize(' ').find {isPalindrome(it)}
}

s1 = "The quick brown fox jumped over anna the dog";
println(findFirstPalindrome(s1))
s2 = "Bob went to Harrah and gambled with Otto and Steve"
println(findFirstPalindrome(s2))
```

例 4-20 的 `isPalindrome()` 方法首先规整目标词的大小写，然后检查词中字符的顺序反转之后，是否还与原来的字符排列相同，也就是所谓的“回文词”（palindrome）。`findFirstPalindrome()` 尝试在一段话里找出第一个回文词，查找过程利用了 Groovy 的 `find()` 方法，筛选的逻辑通过一个闭包传进 `find()`。

假设我们有很长的一段字符序列，需要从里面找出第一个回文词。按照例 4-20 的代码，在执行 `findFirstPalindrome()` 方法的过程中，它首先迫切地对整个序列做词的划分，建立起一个中间数据结构，然后在这个中间结构上开始 `find()` 操作。Groovy 的 `tokenize()` 方法不是缓求值的，当字符序列很长的时候，有可能产生一个十分庞大的临时结构，而其中的大部分数据都会在下一步操作中被丢弃。如果这个例子用 Clojure 语言来实现的话，会怎么样呢？我们来看看例 4-21。

例 4-21　查找“回文词”，Clojure 实现

```
(defn palindrome? [s]
  (let [sl (.toLowerCase s)]
  (= sl (apply str (reverse sl)))))

(defn find-palindromes [s]
  (filter palindrome? (clojure.string/split s #" ")))

(println (find-palindromes "The quick brown fox jumped over anna the dog"))
; (anna)
(println (find-palindromes "Bob went to Harrah and gambled with Otto and Steve"))
;(Bob Harrah Otto)
(println (take 1 (find-palindromes "Bob went to Harrah with Otto and Steve")))
;(Bob)
```

例 4-20 和例 4-21 的实现思路完全一样，只是使用了不同的语言构造。例 4-21 的 `(palindrome? )` 函数首先将参数字符串规整为小写形式，然后检查字符顺序反转之后得到的字符串是否相同。多出来的 `apply` 调用是为了把 `reverse` 返回的字符序列转成 `String` 类型，方便比较。`(find-palindromes )` 函数利用了 Clojure 提供的 `(filter )` 函数，`(filter )` 要求传入充当筛选逻辑的一个函数以及待筛选的集合。按照 Clojure 的语法，表达对 `(palindrome? )` 函数的调用可以有不同的写法。我们可以建立一个匿名函数来完成调用，简写为 `#(palindrome? %)` 的形式。这是利用了 Clojure 的语法便利，略去匿名函数的声明步骤，并省略参数的命名，以 `%` 符号来指代唯一的参数。例 4-21 不必求助于匿名函数，它只要直接写下函数名就可以了；`(filter )` 要求传入一个单参数且返回布尔类型的函数，`(palindrome? )` 符合这个要求。

从 Groovy 翻译成 Clojure 代码，不仅仅发生了语法上的转换。Clojure 的数据结构都是默认缓求值的，包括各种集合操作也是如此，如例中用到的 `filter` 和 `split`。因此在 Clojure 版的实现里，一切都自动地具备缓求值特性，这种性质在例 4-21 的第二则演示中发挥了作用：当我们在含有多个匹配项的集合上调用 `(find-palindromes )` 的时候，从 `(filter )` 返回的是一个缓求值的集合，只是后续的打印操作迫使它落实了求值结果。如果我们只想要

第一个匹配项，那么可以像第三则演示那样，圈定缓求值的结果名额。

Scala 实现缓求值特性的手法略有不同。它没有把一切都默认安排成缓求值的，而是在集合之上另外提供了一层缓求值的视图。我们来看例 4-22 的 Scala 版回文词查找实现。

例 4-22　查找“回文词”，Scala 实现

```
def isPalindrome(x: String) = x == x.reverse
def findPalidrome(s: Seq[String]) = s find isPalindrome

findPalindrome(words take 1000000)
```

按照例 4-22 的写法，`take` 方法从非缓求值集合中取出前一百万个词的操作效率会很差，而当我们的目标仅仅是从中找出第一个回文词的时候，代价和收获就更不相称了。为了让 `words` 变成缓求值的集合，我们调用 `view` 方法：

```
findPalindrome(words.view take 1000000)
```

`view` 方法开启了对集合的缓求值遍历，将大幅提高后续代码的执行效率。

4.2.6　缓求值的字段初始化

在结束缓求值这个话题之前，有必要提一下缓求值特性在 Scala 和 Groovy 语言中的另一处体现。它们都为推迟昂贵的初始化工作提供了便利。Scala 只要在 `val` 声明前面加上“lazy”字样，就可以令字段从严格求值变成按需要求值：

```
lazy val x = timeConsumingAndOrSizableComputation()
```

这种写法其实是一层语法糖衣，相当于下面的代码：

```
var _x = None
def x = if (_x.isDefined) _x.get else {
  _x = Some(timeConsumingAndOrSizableComputation())
  _x.get
}
```

Groovy 也提供了效果差不多的便利语法，不过它是通过一种高级语言特性*抽象语法树*变换来实现的。这种特性允许我们操作编译器产生的内部结构*抽象语法树*（Abstract Syntax Tree，AST），在很基础的层次上实施变换。Groovy 预定义了若干变换，其中就有 `@Lazy` 标注，用法如例 4-23 所示。

例 4-23　Groovy 的缓初始化字段

```
class Person {
    @Lazy pets = ['Cat', 'Dog', 'Bird']
}

def p = new Person()
```

```
assert !(p.dump().contains('Cat'))

assert p.pets.size() == 3
assert p.dump().contains('Cat')
```

例 4-23 的测试说明，`Person` 实例 `p` 一开始并不包含 `Cat` 这个值，直到我们首次访问相关结构的时候，它才被初始化进去。Groovy 还允许用闭包来初始化字段：

```
class Person {
    @Lazy List pets = { /* 各种复杂运算 */ }()
}
```

更进一步，我们可以让 Groovy 通过软引用（soft reference）来持有缓初始化的字段，软引用是 Java 平台上可以按需要回收的一种指针引用：

```
class Person {
    @Lazy(soft = true) List pets = ['Cat', 'Dog', 'Bird']
}
```

最后我们得到内存使用效率最高的一个版本，一边尽量推迟初始化，一边积极地按需回收内存。

第5章

演化的语言

函数式编程语言和面向对象语言对待代码重用的方式不一样。面向对象语言喜欢大量地建立有很多操作的各种数据结构，函数式语言也有很多的操作，但对应的数据结构却很少。面向对象语言鼓励我们建立专门针对某个类的方法，我们从类的关系中发现重复出现的模式并加以重用。函数式语言的重用表现在函数的通用性上，它们鼓励在数据结构上使用各种共通的变换，并通过高阶函数来调整操作以满足具体事项的要求。

软件开发中有一些问题会反复地出现，各种语言为了解决特定的问题而演化出了不同的解决方案，这就是本章要讨论的话题。我们将谈及函数式编程对于自定义数据结构的态度落差，语言语法的可塑性和由此衍生的解答思路，还将谈及分发（dispatch）问题的答案选项，运算符重载，以及函数式数据结构。

5.1 少量的数据结构搭配大量的操作

> 100 个函数操作一种数据结构的组合，要好过 10 个函数操作 10 种数据结构的组合。
>
> ——Alan Perlis

在面向对象的命令式编程语言里面，重用的单元是类和用作类间通信的消息，通常可以表述成一幅类图（class diagram）。例如这个领域的开拓性著作《设计模式：可复用面向对象软件的基础》就给每一个模式都至少绘制了一幅类图。在 OOP 的世界里，开发者被鼓励针对具体的问题建立专门的数据结构，并以方法的形式，将专门的操作关联在数据结构上。函数式编程语言选择了另一种重用思路。它们用很少的一组关键数据结构（如 `list`、`set`、`map`）来搭配专为这些数据结构深度优化过的操作。我们在这些关键数据结构和操作

组成的一套运转机构上面，按需要“插入”另外的数据结构和高阶函数来调整机器，以适应具体的问题。例如我们已经在几种语言中操练过的 `filter()` 函数，传给它的代码块就是这么一个“插入”的部件，筛选的条件由传入的高阶函数确定，而运转机构则负责高效率地实施筛选，并返回筛选后的列表。

比起在定制的类结构上做文章，把封装的单元缩小到函数级别，有利于在更基础的层面上更细粒度地实施重用。Clojure 很好地发挥了这方面的优势，例如在 XML 的解析问题上。Java 语言的 XML 解析框架数量繁多，每一种都有自己的定制数据结构和方法语义（如 SAX 和 DOM 都是自成一体）。Clojure 的做法相反 ，它不鼓励使用专门的数据结构，而是将 XML 解析成标准的 `Map` 结构。而 Clojure 有极为丰富的工具可以与 map 结构相配合，比如我们只需要利用内建的 list-comprehension 函数 `for`，就可以实现 XPath 风格的查询，如例 5-1 所示。

例 5-1　用 Clojure 语言解析 XML

```
(use 'clojure.xml)

(def WEATHER-URI "http://weather.yahooapis.com/forecastrss?w=%d&u=f")

(defn get-location [city-code]
  (for [x (xml-seq (parse (format WEATHER-URI city-code)))
        :when (= :yweather:location (:tag x))]
    (str (:city (:attrs x)) "," (:region (:attrs x)))))

(defn get-temp [city-code]
  (for [x (xml-seq (parse (format WEATHER-URI city-code)))
        :when (= :yweather:condition (:tag x))]
    (:temp (:attrs x))))

(println "weather for " (get-location 12770744) "is " (get-temp 12770744))
```

例 5-1 访问了 Yahoo! 的天气服务来取得给定城市的天气预报。Clojure 作为一种 Lisp 变种，它的代码按照由内而外的顺序阅读起来会容易一些。对服务端口的实际调用发生在 `(parse (format WEATHER-URI city-code))`，这里利用了 `String` 类的 `format()` 函数来向字符串中嵌入 `city-code` 字段。然后 list comprehension 函数 `for` 将经 `xml-seq` 转换后的 XML 解析结果，放入可查询的 map 结构 `x`。接下来由 `:when` 谓词进行匹配，我们要找的是 `:yweather:condition` 标签（已经被转换成 Clojure 关键字）。

为了更好地理解这几行语句是怎么从 map 结构中取出数值的，我们可以先看看结构里到底放了哪些内容。从天气服务返回的内容，解析过后会是下面的样子：

```
({:tag :yweather:condition, :attrs {:text Fair, :code 34, :temp 62, :date Tue,
   04 Dec 2012 9:51 am EST}, :content nil})
```

由于 Clojure 特别为操作 map 结构而做的优化，map 结构中的关键字同时也是该结构的一个函数。例 5-1 中调用 `(:tag x)`，意思相当于“从保存在 `x` 的 map 中取出 `:tag` 键所对应

的值”。同理，通过 `:yweather:condition` 也可以取出它所对应的值，也就是另一个 map 结构 `attrs`。随后我们又用相同的语法，从 `attrs` 中取出了 `:temp` 键所对应的值。

Clojure 似乎有无数种方法可以操作 map 和其他核心数据结构，令初学者望而生畏。这种状况反映了一个事实，Clojure 的大部分特性都是围绕这些核心的、高度优化的数据结构而存在的。与其另外搭建一套框架来容纳 XML 的解析结果，Clojure 选择将之适配到已有的核心结构上，因为这边已经准备好了一大套现成的工具。

Clojure 的 XML 库就从这种依托于基础数据结构的思路中得到了好处。有一种为遍历树形结构（如 XML 文档）而设计的数据结构叫作 zipper，由 Gérard Huet 在 1997 年提出。zipper 结构让我们用坐标方向来指示在树结构中移动的步骤。例如从树的根部开始，我们可以发出 `(-> z/down z/down z/right)` 命令来移动到第二层的右元素。Clojure 已经为我们准备了将 XML 解析结果转换成 zipper 的函数，让各种树形结构都能够以统一方式来完成遍历。

5.2 让语言去迎合问题

很多开发者都有一种误解，认为自己的工作就是把复杂的业务问题翻译成某种编程语言，如 Java。他们会有这样的想法，原因在 Java 身上。Java 不是一种特别灵活的语言，我们只能死板地用一些现成结构来拼凑自己的设计。而另外一些开发者使用的语言可塑性更强，他们不会拿问题去硬套语言，而是想法揉捏手中的语言来迎合问题。不少语言显示出了这方面的潜力，例如 Ruby 对领域专用语言（DSL）的支持就比主流语言要强得多。现代的函数式语言在这方面走得更远。Scala 从设计之初就为充当内部 DSL 的宿主做好了准备。另外所有 Lisp 家族的语言（包括 Clojure）都传承了无可比拟的灵活性，可以任由开发者根据问题重塑语言。我们来看例 5-2，它利用 Scala 提供的 XML 基本功能重新实现了例 5-1 的天气查询示例。

例 5-2　Scala 语言为操作 XML 准备的语法糖衣

```
import scala.xml._
import java.net._
import scala.io.Source

val theUrl = "http://weather.yahooapis.com/forecastrss?w=12770744&u=f"

val xmlString = Source.fromURL(new URL(theUrl)).mkString
val xml = XML.loadString(xmlString)
val city = xml \\ "location" \\ "@city"
val state = xml \\ "location" \\ "@region"
val temperature = xml \\ "condition" \\ "@temp"

println(city + ", " + state + " " + temperature)
```

Scala 语言从设计上就考虑了可塑性，它允许我们使用运算符重载（本章稍后详细讨论）、

隐含类型等手段来扩展语言。例 5-2 用来实现 XPath 式查询的 \\ 运算符，就是 Scala 语言上的一个扩展。

重塑语言的能力算不上函数式语言独有的特性，现代语言普遍可以轻巧地揉捏语言来贴合问题域，不过在这种能力的影响下，更容易催生带有浓厚函数式、描述式风格的代码。

让程序去贴合问题，不要反过来。

5.3 对分发机制的再思考

第 3 章介绍过的 Scala 的模式匹配特性，就是一种分发机制，我们用"分发机制"这个词来泛称各种语言中用作"动态地选择行为"的特性。与 Java 的做法相比，几种函数式 JVM 语言的分发机制更加简洁、灵活，我们就用这一节来讨论它们。

5.3.1 Groovy对分发机制的改进

Java 代码要表述"条件执行"，除了很少的一些情况适用 `switch` 语句，大多离不开 `if` 语句。由于长串的 `if` 语句难以阅读，Java 开发者通常需要依赖 GoF 模式集里面的 Factory 模式（或者 Abstract Factory 模式）来缓解问题。如果语言直接提供更灵活的方式来表述复杂的判断，我们就不必负担随模式而来的额外的结构，于是代码就大大简化了。

Groovy 的 `switch` 语句在语法上模仿 Java，但功能要比 Java 的 `switch` 语句强大很多，如例 5-3 所示。

例 5-3　Groovy 语言大幅改进过的 `switch` 语句

```
package com.nealford.ft.polydispatch

class LetterGrade {
  def gradeFromScore(score) {
    switch (score) {
      case 90..100 : return "A"
      case 80..<90 : return "B"
      case 70..<80 : return "C"
      case 60..<70 : return "D"
      case 0..<60 : return "F"
      case ~"[ABCDFabcdf]" : return score.toUpperCase()
      default: throw new IllegalArgumentException("Invalid score: ${score}")
    }
  }
}
```

例 5-3 按照 `score` 的取值返回相应的字母来代表成绩等级。Groovy 的 `switch` 语句允许使用宽泛的动态类型，没有 Java 的类型限制。例 5-3 的 `score` 参数可以是从 0 到 100 的数字，也可以是字母等级。Groovy 的 `switch` 也和 Java 一样，遵循相同的“fall-through”语义，如果没有用 `return` 或 `break` 来终结一则 `case`，就会继续“跌落”到下一则。但 Groovy 的 `case` 条件不像 Java 那么死板，我们可以指定区间（`90..100`）、开区间（`80..<90`）、正则表达式（`~"[ABCDFabcdf]"`），最后写上兜底的 `default` 条件。

由于 Groovy 语言的动态类型特质，我们可以传入不同类型的参数并分别给予恰当的处理，如例 5-4 的单元测试所示。

例 5-4 测试 Groovy 版的成绩分等程序

```
import org.junit.Test
import com.nealford.ft.polydispatch.LetterGrade

import static org.junit.Assert.assertEquals

class LetterGradeTest {
  @Test
  public void test_letter_grades() {
    def lg = new LetterGrade()
    assertEquals("A", lg.gradeFromScore(92))
    assertEquals("B", lg.gradeFromScore(85))
    assertEquals("D", lg.gradeFromScore(65))
    assertEquals("F", lg.gradeFromScore("f"))
  }
}
```

加强的 `switch` 在连串 `if` 和 Factory 设计模式之间提供了一个有用的平衡点。Groovy 的 `switch` 允许匹配区间和其他复杂类型，可在编程中起到与 Scala 模式匹配类似的作用。

5.3.2 “身段柔软”的Clojure语言

Java 以及很多类似的语言都包含“关键字”的概念，把它们当作语法上的支点。在这类语言中，开发者不可以自创新的关键字（不过有的语言允许通过元编程来实现语言扩展），关键字蕴含了开发者无法从其他地方获得的语义。例如 Java 的 `if` 语句懂得对布尔运算作短路处理。我们可以在 Java 语言下创建函数和类，但没办法创造基本的语法构造单元，因此必须把问题翻译成符合编程语言语法的陈述。（实际上很多开发者认为自己的工作就是从事这种翻译活动。）而在 Clojure 等 Lisp 家族的语言下，开发者可以根据问题来修改语言，并没有一条明确的界线隔开语言设计者和使用语言来进行创作的开发者。

Clojure 可以写出富于可读性的（Lisp 风格的）代码。例 5-5 用 Clojure 语言重新实现了前面的成绩分等例子。

例 5-5　成绩分等的 Clojure 实现

```
(ns lettergrades)

(defn in [score low high]
  (and (number? score) (<= low score high)))

(defn letter-grade [score]
  (cond
    (in score 90 100) "A"
    (in score 80 90) "B"
    (in score 70 80) "C"
    (in score 60 70) "D"
    (in score 0 60) "F"
    (re-find #"[ABCDFabcdf]" score) (.toUpperCase score)))
```

例 5-5 完成分等工作的 `letter-grade` 函数看上去一目了然，这要归功于我们实现的 `in` 函数。我们用 `cond` 函数来执行一系列测试，`in` 函数负责判断测试条件。这个版本也和前面的一样，允许打分数字和等级字母。最后的返回结果要求是大写字母，因此我们对最后的返回字符串调用 `toUpperCase` 函数，将万一传入的小写字母转换成大写。在 Clojure 语言里，函数是比类更优先的语言成分，它的函数调用用 Java 的眼光来看，就像内外颠倒了一样：Java 下的 `score.toUpperCase()` 调用，在 Clojure 下的等价写法是 `(.toUpperCase score)`。

我们来测试一下 Clojure 版的成绩分等程序，请看例 5-6。

例 5-6　测试 Clojure 版的成绩分等程序

```
(ns nealford-test
  (:use clojure.test)
  (:use lettergrades))

(deftest numeric-letter-grades
  (dorun (map #(is (= "A" (letter-grade %))) (range 90 100)))
  (dorun (map #(is (= "B" (letter-grade %))) (range 80 89)))
  (dorun (map #(is (= "C" (letter-grade %))) (range 70 79)))
  (dorun (map #(is (= "D" (letter-grade %))) (range 60 69)))
  (dorun (map #(is (= "F" (letter-grade %))) (range 0 59))))

(deftest string-letter-grades
  (dorun (map #(is (= (.toUpperCase %)
           (letter-grade %))) ["A" "B" "C" "D" "F" "a" "b" "c" "d" "f"])))

(run-all-tests)
```

这个单元测试比实现本身还要复杂！但不管怎么说，代码很好地体现了 Clojure 语言简洁的特点。

`numeric-letter-grades` 测试希望把每个等级区间内的全部数值都验证一遍。我们按照从内到外的顺序理解代码，`#(is (= "A" (letter-grade %)))` 的工作是创建一个匿名函数，如

果得到的等级字母正确，就返回 `true`。再往外一层，`map` 函数将匿名函数映射到第二参数位置上指定的集合，即对应区间内的数值列表上。

`(dorun )` 函数会迫使副作用生效，我们的测试框架依赖于这一点。例 5-6 在每个区间上执行的 `map` 操作，都应该产生一个全是 `true` 值的列表。来自 `clojure.test` 命名空间的 `(is )` 函数会在其副作用中逐一检验这些返回值。我们把映射函数放在 `(dorun )` 里执行，才能正确地令副作用生效，达到测试目的。

5.3.3 Clojure的多重方法和基于任意特征的多态

长串的 `if` 语句难以阅读和查错，可是 Java 在语言层面找不到好一点的替代品，一般只能通过 GoF 设计模式集里面的 Factory 模式和 Abstract Factory 模式来缓解问题。Factory 模式的运作机制利用了 Java 语言基于类的多态，我们在父类或接口中定义共通的方法签名，然后动态地选择要执行的具体实现。

很多开发者因为 Clojure 不是一种面向对象语言而排斥它，盲目地认为面向对象语言才是能力最强的语言。这是错误的想法。Clojure 拥有面向对象语言的一切特性，不需要依靠其他特性来间接模拟。例如多态，不但是 Clojure 直接支持的一种特性，而且不必局限于按类来判断分发。Clojure 承载多态语义的*多重方法*（multimethod）特性允许开发者使用任意特征（及其组合）来触发分发。

Clojure 习惯用 `struct` 来放置数据，`struct` 有点像一个只有数据部分的类。请看例 5-7 的 Clojure 代码。

例 5-7 用 Clojure 定义一个表示色彩的数据结构

```
(defstruct color :red :green :blue)

(defn red [v]
  (struct color v 0 0))

(defn green [v]
  (struct color 0 v 0))

(defn blue [v]
  (struct color 0 0 v))
```

例 5-7 首先定义了包含三个分量值的结构来表示颜色。另外还定义了三个方法，各返回饱和度可调的一种单色。

Clojure 的*多重方法*是一种特别的方法定义形式，它的参数是一个返回判断条件的分发函数。后续定义的一系列同名方法分别对应到不同的分发条件。例 5-8 给出了定义多重方法的一个实例。

例 5-8　定义一个多重方法

```
(defn basic-colors-in [color]
  (for [[k v] color :when (not= v 0)] k))

(defmulti color-string basic-colors-in)

(defmethod color-string [:red] [color]
  (str "Red: " (:red color)))

(defmethod color-string [:green] [color]
  (str "Green: " (:green color)))

(defmethod color-string [:blue] [color]
  (str "Blue: " (:blue color)))

(defmethod color-string :default [color]
  (str "Red:" (:red color) ", Green: " (:green color) ", Blue: " (:blue color)))
```

例 5-8 首先定义了 `basic-colors-in` 分发函数，它用一个 vector 结构返回所有非零的颜色分量。在方法的各实现版本中，我们针对分发函数的返回值是一种单色的情况做了特别处理；例中返回了一个标示颜色的字符串。我们给最后一个版本加上了可选的 `:default` 关键字，让它负责所有未作特殊处理的情况。这个方法收到的颜色值参数不再是单色的，因此返回时要列举所有的颜色分量。

我们来对这组多重方法做一些测试，请看例 5-9。

例 5-9　测试 Clojure 版的色彩模型

```
(ns color-dispatch.core-test
  (:require [clojure.test :refer :all]
            [color-dispatch.core :refer :all]))

(deftest pure-colors
  (is (= "Red: 5" (color-string (struct color 5 0 0))))
  (is (= "Green: 12" (color-string (struct color 0 12 0))))
  (is (= "Blue: 40" (color-string (struct color 0 0 40)))))

(deftest varied-colors
  (is (= "Red:5, Green: 40, Blue: 6" (color-string (struct color 5 40 6)))))

(run-all-tests)
```

例 5-9 的测试表明，如果调用时传入的参数是单色，该多重方法将执行对应的单色版本。如果我们传入复合的颜色，则会触发默认方法，由它返回所有的颜色分量值。

切断多态和继承之间的耦合关系，催生了一种强大的、灵活周全的分发机制。这样的分发机制能够处理相当复杂的情况，例如不同图像文件格式的分发问题，每种格式类型都是由各不相同的一组特征来定义的。多重方法赋予了 Clojure 构造强大分发机制的能力，其适应性不输于 Java 的多态，而且限制更少。

5.4 运算符重载

运算符重载是函数式语言常见的特性，它允许我们重新定义运算符（诸如 +、-、*），使之适用于新的类型，并承载新的行为。Java 语言没有运算符重载特性，这是它的设计者在语言形成阶段就刻意作出的决定，不过时至今日，几乎所有现代语言都包含了运算符重载特性，连同 Java 平台上的一众衍生语言在内。

5.4.1 Groovy

Groovy 希望既改造 Java 的语法，同时又自然地保留 Java 的语义。在此思路下，Groovy 将运算符自动映射成方法，从而令运算符重载变成了方法的实现问题。例如我们想针对 `Integer` 类重载 + 运算符的话，只要覆盖该类的 `plus()` 方法即可。完整的映射列表可以查阅 Groovy 的文档，表 5-1 列出了几种常用运算符的映射关系。

表5-1：Groovy语言部分运算符和方法之间的映射关系

运算符	方法
`x + y`	`x.plus(y)`
`x * y`	`x.multiply(y)`
`x / y`	`x.div(y)`
`x ** y`	`x.power(y)`

为了演示运算符重载，我们试着用 Groovy 和 Scala 分别定义一个表示复数的 `ComplexNumber` 类。复数是一个数学概念，它由实部和虚部两部分组成，例如“3 + 4i”。复数在各种学科如工程学、物理学、电磁学、混沌理论等领域中有着广泛的应用。

假如能够在表述问题时直接使用专业领域的运算符，将会极大地方便这些领域的开发者。

例 5-10 用 Groovy 实现了一个复数类 `ComplexNumber`。

例 5-10　Groovy 版的复数模型 `ComplexNumber`

```
package complexnums

class ComplexNumber {
   def real, imaginary

  public ComplexNumber(real, imaginary) {
    this.real = real
    this.imaginary = imaginary
  }

  def plus(rhs) {
    new ComplexNumber(this.real + rhs.real, this.imaginary + rhs.imaginary)
  }
```

```
    // 公式:(x + yi)(u + vi) = (xu - yv) + (xv + yu)i.
    def multiply(rhs) {
      new ComplexNumber(
          real * rhs.real - imaginary * rhs.imaginary,
          real * rhs.imaginary + imaginary * rhs.real)
    }

    def String toString() {
      real.toString() + ((imaginary < 0 ? "" : "+") + imaginary + "i").toString()
    }
  }
```

例 5-10 在类中定义了分别代表实部和虚部的属性，并为运算符重载实现了 `plus()` 和 `multiply()` 方法。两个复数的加法运算比较简单：`plus()` 运算符将两数的实部相加作为和的实部，虚部相加作为和的虚部。两个复数的乘法要按照下面的公式来计算：

```
(x + yi)(u + vi) = (xu - yv) + (xv + yu)i
```

例 5-10 的 `multiply()` 运算符实现完全照公式进行。两数实部的乘积减去虚部的乘积，差作为结果的实部；两数的实部与虚部交换相乘，两积之和作为结果的虚部。

我们来测试一下刚刚定义的复数运算符，请看例 5-11。

例 5-11　测试复数运算符

```
package complexnums

import org.junit.Test
import static org.junit.Assert.assertTrue
import org.junit.Before

class ComplexNumberTest {
  def x, y

  @Before void setup() {
    x = new ComplexNumber(3, 2)
    y = new ComplexNumber(1, 4)
  }

  @Test void plus() {
    def z = x + y;
    assertTrue 3 + 1 == z.real
    assertTrue 2 + 4 == z.imaginary
  }

  @Test void multiply() {
    def z = x * y
    assertTrue(-5 == z.real)
    assertTrue 14 == z.imaginary
  }

  @Test void to_string() {
    assertTrue "3+2i" == x.toString()
```

```
      assertTrue "4+6i" == (x + y).toString()
      assertTrue "3+0i" == new ComplexNumber(3, 0).toString()
      assertTrue "4-2i" == new ComplexNumber(4, -2).toString()
    }
  }
```

例 5-11 在 `plus()` 和 `multiply()` 两个测试方法中使用了重载后的运算符，算式里的符号完全是领域专家习惯的写法，而且和内建类型的算式看不出区别。

5.4.2 Scala

Scala 也支持运算符重载，它的做法是完全不区分运算符和方法：运算符不过是一些名字比较特别的方法罢了。因此，如果我们想覆盖乘法运算符的默认行为，只要实现 `*` 方法就可以了。例 5-12 用 Scala 实现了一个复数类。

例 5-12　Scala 版的复数模型

```
final class Complex(val real: Int, val imaginary: Int) extends Ordered[Complex] {

  def +(operand: Complex) =
      new Complex(real + operand.real, imaginary + operand.imaginary)

  def +(operand: Int) =
    new Complex(real + operand, imaginary)

  def -(operand: Complex) =
    new Complex(real - operand.real, imaginary - operand.imaginary)

  def -(operand: Int) =
    new Complex(real - operand, imaginary)

  def *(operand: Complex) =
      new Complex(real * operand.real - imaginary * operand.imaginary,
          real * operand.imaginary + imaginary * operand.real)

  override def toString() =
      real + (if (imaginary < 0) "" else "+") + imaginary + "i"

  override def equals(that: Any) = that match {
    case other : Complex => (real == other.real) && (imaginary == other.imaginary)
    case other : Int => (real == other) && (imaginary == 0)
    case _ => false
  }

  override def hashCode(): Int =
    41 * ((41 + real) + imaginary)

  def compare(that: Complex) : Int = {
    def myMagnitude = Math.sqrt(real ^ 2 + imaginary ^ 2)
    def thatMagnitude = Math.sqrt(that.real ^ 2 + that.imaginary ^ 2)
    (myMagnitude - thatMagnitude).round.toInt
  }
}
```

例 5-12 在类中定义了表示实部和虚部的成员，并实现了加法、减法和乘法运算的运算符 / 方法。按照 Scala 的语法，默认构造器的参数直接写在类名的后面，我们从构造器传入 `real` 和 `imaginary` 参数作为复数的实部和虚部。Scala 会自动令默认构造器的参数成为类字段，因此我们看到类中不必写出字段的定义，只需要定义方法就可以了。我们定义了以加法、减法和乘法符号命名的若干方法，允许传入 `Complex` 类型的参数。

例 5-12 中 `toString()` 方法的实现方式反映了函数式语言共同的一点小习惯：可以用表达式（expression）的地方就不用语句（statement）。`toString()` 方法在虚部为正的情况下，必须在虚部前面添加 + 符号，实部的符号则不需要特别处理。`if` 在 Scala 语言里不是语句而是表达式，Scala 语言也因此不需要像 Java 那样的三元运算符（`?:`）。

现在我们可以像内建数字类型那样，使用各种运算符来书写复数的算式，如例 5-13 所示。

例 5-13　测试 Scala 版的复数实现

```
import org.scalatest.FunSuite

class ComplexTest extends FunSuite {

  def fixture =
    new {
        val a = new Complex(1, 2)
        val b = new Complex(30, 40)
    }

  test("plus") {
    val f = fixture
    val z = f.a + f.b
    assert(1 + 30 == z.real)
  }

  test("comparison") {
    val f = fixture
    assert(f.a < f.b)
    assert(new Complex(1, 2) <= new Complex(3, 4))
    assert(new Complex(1, 1) < new Complex(2,2))
    assert(new Complex(-10, -10) > new Complex(1, 1))
    assert(new Complex(1, 2) >= new Complex(1, 2))
    assert(new Complex(1, 2) <= new Complex(1, 2))
  }
}
```

Java 语言的设计者从使用 C++ 语言的经验中得出结论，认为运算符重载会给语言增加过多的复杂性，因此刻意从 Java 语言里排除了这种特性。现代语言大多已经相当程度地消除了定义上的复杂性，但以往关于滥用运算符重载的告诫都还是成立的。

要想契合问题域的表达习惯，可以利用运算符重载来改变语言的外貌，不必创造全新的语言。

5.5 函数式的数据结构

Java 语言习惯使用异常来处理错误，语言本身提供了异常的创建和传播机制。假如语言中不存在结构性的异常处理机制，我们应该怎样处理错误呢？很多函数式语言根本就没有 Java 那样的“异常”概念，它们肯定有别的什么方式可以表达错误状况下的行为。

“异常”违背了大多数函数式语言所遵循的一些前提条件。首先，函数式语言偏好没有副作用的纯函数。抛出异常的行为本身就是一种副作用，会导致程序路径偏离正轨（进入异常的流程）。函数式语言以操作值为其根本，因此喜欢在返回值里表明错误并作出响应，这样就不需要打断程序的一般流程了。

引用的透明性（referential transparency）是函数式语言重视的另一项性质：发出调用的例程不必关心它的访问对象真的是一个值，还是一个返回值的函数。可是如果函数有可能抛出异常的话，用它来代替值就不再是安全的了。

在这一节里，我们将参考 Functional Java 框架的一些做法，使用 Java 语言，但完全抛开通常的异常传播机制来实现一种类型安全的错误处理范式。

5.5.1 函数式的错误处理

我们在 Java 语言下抛开异常来处理错误，返回值的限制是首先会遇到的绊脚石，语言规定了方法只能返回一个值。不过，我们当然可以把多个值装进单个 `Object`（或其子类）对象里面再一起返回，这样就不违反规定了。例如像例 5-14 的 `divide()` 方法一样，利用 `Map` 来返回多个值。

例 5-14　利用 `Map` 来返回多个值

```
public static Map<String, Object> divide(int x, int y) {
    Map<String, Object> result = new HashMap<String, Object>();
    if (y == 0)
        result.put("exception", new Exception("被零除"));
    else
        result.put("answer", (double) x / y);
    return result;
}
```

例 5-14 创建了一个键为 `String` 类型、值为 `Object` 类型的 `Map` 结构来装载返回值。在 `divide()` 方法中，我们将键名设为 `exception` 来表示失败，设为 `answer` 则表示成功。例

5-15 对两种情况都做了测试。

例 5-15　测试通过 `Map` 结构返回的成功结果和失败结果

```
@Test
public void maps_success() {
    Map<String, Object> result = RomanNumeralParser.divide(4, 2);
    assertEquals(2.0, (Double) result.get("answer"), 0.1);
}

@Test
public void maps_failure() {
    Map<String, Object> result = RomanNumeralParser.divide(4, 0);
    assertEquals("被零除", ((Exception) result.get("exception")).getMessage());
}
```

例 5-15 的 `maps_success` 测试从 `Map` 中取出了正确的结果，`maps_failure` 测试也正确地接收到了异常报告。

用 `Map` 来返回多个值的设计存在一些明显的缺点。首先，`Map` 中放置的内容没有类型安全的保障，编译器无法捕捉到类型方面的错误。假如改用枚举类型来充当键，可以稍微弥补这个缺点，但效果有限。第二，方法的调用者无法直接得知执行是否成功，需要逐一比对所有可能键，给调用者带来负担。第三，没有办法强制结果只含有一对键值，届时将出现歧义。

我们真正需要的机制，不但要返回两个（或更多）值，还必须能够保证类型安全。

5.5.2　Either类

函数式语言也经常会遇到返回两种截然不同的值的需求，它们用来建模这种行为的常用数据结构是 `Either` 类。`Either` 的设计规定了它要么持有“左值”，要么持有“右值”，但绝不会同时持有两者。这种数据结构也被称为*不相交联合体*（disjoint union）。C 语言和一些衍生语言中有一种*联合体*（union）数据类型，能够在同一个位置上容纳不同类型的单个实例。不相交联合体为两种类型的实例都准备了位置，但只会持有其中一种类型的单个实例。

Scala 语言提供了 `Either` 类的实现，例 5-16 演示了它的用法。

例 5-16　Scala 自带的 `Either` 类

```
type Error = String
type Success = String
def call(url:String):Either[Error,Success]={
    val response = WS.url(url).get.value.get
    if (valid(response))
        Right(response.body)
    else Left("Invalid response")
}
```

如例 5-16 所示，错误处理是 Either 的主要用途之一。Either 类很好地融入了 Scala 语言的大环境，与其他部件配合无间，如例 5-17 演示了在 Either 上进行的模式匹配。

例 5-17　Scala 的 Either 类和模式匹配

```
getContent(new URL("http://nealford.com")) match {
  case Left(msg) => println(msg)
  case Right(source) => source.getLines.foreach(println)
}
```

虽然 Java 语言没有内建这种类型，但我们可以利用泛型来制作一个替代品。例 5-18 在 Java 语言下完成了一个简单的 Either 类实现。

例 5-18　通过 Either 类（类型安全地）返回两种值

```
package com.nealford.ft.errorhandling;

public class Either<A,B> {
    private A left = null;
    private B right = null;

    private Either(A a,B b) {
        left = a;
        right = b;
    }

    public static <A,B> Either<A,B> left(A a) {
        return new Either<A,B>(a,null);
    }

    public A left() {
        return left;
    }

    public boolean isLeft() {
        return left != null;
    }

    public boolean isRight() {
        return right != null;
    }

    public B right() {
        return right;
    }

    public static <A,B> Either<A,B> right(B b) {
        return new Either<A,B>(null,b);
    }

    public void fold(F<A> leftOption, F<B> rightOption) {
        if(right == null)
            leftOption.f(left);
        else
```

```
            rightOption.f(right);
        }
    }
```

例 5-18 中 Either 类的构造器是私有的，两个静态方法 left(A a) 和 right(B b) 承担了对外的构造责任。类中余下的方法都是用来获取和确认类成员的辅助方法。

有了 Either 这件利器，我们的代码就可以在确保类型安全的前提下，视情况返回异常或者有效结果（但不会同时返回两者）。按照函数式编程的传统习惯，异常（如果有的话）置于 Either 类的*左值*上，正常结果则放在*右值*。

1. 解析罗马数字

我们用一个解析罗马数字的例子来演示 Either 的用法，例中需要用到一个表示罗马数字的类 RomanNumeral，如例 5-19 所示。

例 5-19　罗马数字的简单实现，Java 语言

```
package com.nealford.ft.errorhandling;

public class RomanNumeral {

    private static final String NUMERAL_MUST_BE_POSITIVE =
            "Value of RomanNumeral must be positive.";
    private static final String NUMERAL_MUST_BE_3999_OR_LESS =
            "Value of RomanNumeral must be 3999 or less.";
    private static final String DOES_NOT_DEFINE_A_ROMAN_NUMERAL =
            "An empty string does not define a Roman numeral.";
    private static final String NO_NEGATIVE_ROMAN_NUMERALS =
            "Negative numbers not allowed";
    private static final String NUMBER_FORMAT_EXCEPTION =
            "Illegal character in Roman numeral.";

    private final int num;

    private static int[] numbers = {1000, 900, 500, 400, 100, 90,
            50, 40, 10, 9, 5, 4, 1};
    private static String[] letters = {"M", "CM", "D", "CD", "C", "XC",
            "L", "XL", "X", "IX", "V", "IV", "I"};

    public RomanNumeral(int arabic) {
        if (arabic < 1)
            throw new NumberFormatException(NUMERAL_MUST_BE_POSITIVE);
        if (arabic > 3999)
            throw new NumberFormatException(NUMERAL_MUST_BE_3999_OR_LESS);
        num = arabic;
    }

    public RomanNumeral(String roman) {
        if (roman.length() == 0)
            throw new NumberFormatException(DOES_NOT_DEFINE_A_ROMAN_NUMERAL);
        if (roman.charAt(0) == '-')
            throw new NumberFormatException(NO_NEGATIVE_ROMAN_NUMERALS);
```

```
        roman = roman.toUpperCase();

        int positionInString = 0;
        int arabicEquivalent = 0;

        while (positionInString < roman.length()) {
            char letter = roman.charAt(positionInString);
            int number = letterToNumber(letter);
            if (number < 0)
                throw new NumberFormatException(NUMBER_FORMAT_EXCEPTION);
            positionInString++;
            if (positionInString == roman.length())
                arabicEquivalent += number;
            else {
                int nextNumber = letterToNumber(roman.charAt(positionInString));
                if (nextNumber > number) {
                    arabicEquivalent += (nextNumber - number);
                    positionInString++;
                } else
                    arabicEquivalent += number;
            }
        }

        if (arabicEquivalent > 3999)
            throw new NumberFormatException(NUMERAL_MUST_BE_3999_OR_LESS);
        num = arabicEquivalent;
    }

    private int letterToNumber(char letter) {
        switch (letter) {
            case 'I':
                return 1;
            case 'V':
                return 5;
            case 'X':
                return 10;
            case 'L':
                return 50;
            case 'C':
                return 100;
            case 'D':
                return 500;
            case 'M':
                return 1000;
            default:
                return -1;
        }
    }

    public String toString() {
        String romanNumeral = "";
        int remainingPartToConvert = num;
        for (int i = 0; i < numbers.length; i++) {
            while (remainingPartToConvert >= numbers[i]) {
```

```
                romanNumeral += letters[i];
                remainingPartToConvert -= numbers[i];
            }
        }
        return romanNumeral;
    }

    public int toInt() {
        return num;
    }
}
```

我们还需要一个负责调用 RomanNumeral 类来完成解析的 RomanNumeralParser。其中的 parseNumber() 方法如例 5-20 所示。

例 5-20　解析罗马数字

```
public static Either<Exception, Integer> parseNumber(String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return Either.left(new Exception("Invalid Roman numeral"));
    else
        return Either.right(new RomanNumeral(s).toInt());
}
```

我们可以做几个单元测试来验证一下解析结果，如例 5-21 所示。

例 5-21　测试罗马数字解析程序

```
@Test
public void parsing_success() {
    Either<Exception, Integer> result = RomanNumeralParser.parseNumber("XLII");
    assertEquals(Integer.valueOf(42), result.right());
}

@Test
public void parsing_failure() {
    Either<Exception, Integer> result = RomanNumeralParser.parseNumber("FOO");
    assertEquals(INVALID_ROMAN_NUMERAL, result.left().getMessage());
}
```

例 5-21 对 parseNumber() 方法做了极粗略的的验证，远不足以纠举实现中的错误，但我们可以从测试里看出来，解析器确实把错误状况放在了 Either 的左值上，正常结果则放在右值上。两种情况都测试了。

Either 类比起到处传递 Map 结构的方案有了很大的进步。我们保证了类型安全（注意异常的类型还可以按需要细化）；方法声明中明确指出了可能发生的错误，这是对返回值 Either 做泛型宣告时的额外收获；返回结果（无论异常还是正常结果）需从 Either 中取出，多了一道间接层。恰恰是这多出来的间接层，给了我们实现缓求值的空间。

2. 罗马数字解析的缓求值实现和Functional Java框架

Either 类是函数式世界里的熟面孔，许多函数式算法的实现中都可以找到它的身影。自

然地，Functional Java 框架也提供了一个 `Either` 类的实现，我们完全可以用它直接替换例 5-18 和例 5-20 的代码。不过，当 `Either` 和 Functional Java 的其他部件一起配合使用，会表现得更好。例如配合 Functional Java 的 `P1` 类，我们可以实现缓求值的异常处理。

Functional Java 框架下的 `P1` 类纯粹是一个包装类，内含无参数的单个方法 `_1()`。（另外还有 `P2`、`P3` 等类，分别包装了数量不等的方法。）`P1` 作为一个代码块被传递但并不立即执行，我们可以选择适当的场合再执行里面的代码，实际上就是高阶函数的替代品。

Java 的异常在我们抛出异常的时刻就完成了初始化。但如果我们把抛出异常的代码放在一个缓求值的方法中返回，就可以推迟创建异常对象。请看例 5-22 的演示和后面的测试。

例 5-22　使用 Functional Java 框架创建缓求值的解析器

```
public static P1<Either<Exception, Integer>> parseNumberLazy(final String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return new P1<Either<Exception, Integer>>() {
            public Either<Exception, Integer> _1() {
                return Either.left(new Exception("Invalid Roman numeral"));
            }
        };
    else
        return new P1<Either<Exception, Integer>>() {
            public Either<Exception, Integer> _1() {
                return Either.right(new RomanNumeral(s).toInt());
            }
        };
}
```

例 5-23 测试了上面的缓求值实现。

例 5-23　测试基于 Functional Java 框架的缓求值解析器

```
@Test
public void parse_lazy() {
    P1<Either<Exception, Integer>> result =
            RomanNumeralParser.parseNumberLazy("XLII");
    assertEquals(42, result._1().right().intValue());
}

@Test
public void parse_lazy_exception() {
    P1<Either<Exception, Integer>> result =
            RomanNumeralParser.parseNumberLazy("FOO");
    assertTrue(result._1().isLeft());
    assertEquals(INVALID_ROMAN_NUMERAL, result._1().left().getMessage());
}
```

例 5-22 的代码与例 5-20 差异不大，只多了一层 `P1` 包装。在 `parse_lazy` 测试中，我们必须多做一次取出结果的动作，也就是在结果上调用一次 `_1()`，然后才能拿到 `Either` 的*右值*，最终取得我们想要的结果。`parse_lazy_exception` 测试先检查了*左值*是否存在，然后取出异常对象，查看它携带的消息。

测试中异常对象（以及代价昂贵的栈跟踪信息）一直引而不发，直到我们调用 _1() 方法取出 Either 左值的时候，才被创建出来。也就是说，这是一个缓求值的异常，我们可以按需要推迟执行它的构造方法。

3. 提供默认值

缓求值并非使用 Either 类来处理异常的唯一优点，Either 还可以用来提供默认值。请看例 5-24 的演示。

例 5-24　提供合理的默认返回值

```
private static final int MIN = 0;
private static final int MAX = 1000;

    public static Either<Exception, Integer> parseNumberDefaults(final String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return Either.left(new Exception("Invalid Roman numeral"));
    else {
        int number = new RomanNumeral(s).toInt();
        return Either.right(new RomanNumeral(number >= MAX ? MAX : number).toInt());
    }
}
```

例 5-25 的测试展示了默认值的设置效果。

例 5-25　测试默认值

```
@Test
public void parse_defaults_normal() {
    Either<Exception, Integer> result =
        RomanNumeralParser.parseNumberDefaults("XLII");
    assertEquals(42, result.right().intValue());
}
@Test
public void parse_defaults_triggered() {
    Either<Exception, Integer> result =
        RomanNumeralParser.parseNumberDefaults("MM");
    assertEquals(1000, result.right().intValue());
}
```

例 5-25 假设我们不允许出现大于 MAX 的罗马数字，超过上限的数字一律解析成默认值 MAX。parseNumberDefaults() 方法没有忘记把默认值放在 Either 的*右值*上。

4. 用Either来包装异常

Either 类还可以用来包装异常，将结构化的异常处理机制转化成函数式风格，如例 5-26 所示。

例 5-26　用 Either 包装捕获的异常

```
public static Either<Exception, Integer> divide(int x, int y) {
    try {
        return Either.right(x / y);
```

```
        } catch (Exception e) {
            return Either.left(e);
        }
    }
```

我们在例 5-27 中测试这些经过包装的异常。

例 5-27　测试经 Either 包装的异常

```
@Test
public void catching_other_people_exceptions() {
    Either<Exception, Integer> result = FjRomanNumeralParser.divide(4, 2);
    assertEquals((long) 2, (long) result.right().value());
    Either<Exception, Integer> failure = FjRomanNumeralParser.divide(4, 0);
    assertEquals("/ by zero", failure.left().value().getMessage());
}
```

例 5-26 尝试进行除法运算，期间有可能发生 `ArithmeticException` 异常。如果发生异常，我们就把它装进 `Either` 的*左值*，否则通过 `Either` 的*右值*返回运算结果。在 `Either` 的帮助下，我们将传统的异常（包括 checked 异常）修饰成函数式的处理风格。

当然，我们还可以给从被调用方法中抛出的异常增加一层缓求值的包装，如例 5-28 所示。

例 5-28　对捕获到的异常做缓求值包装

```
public static P1<Either<Exception, Integer>> divideLazily(final int x, final int y) {
    return new P1<Either<Exception, Integer>>() {
        public Either<Exception, Integer> _1() {
            try {
                return Either.right(x / y);
            } catch (Exception e) {
                return Either.left(e);
            }
        }
    };
}
```

例 5-29 对经过缓求值包装的捕获异常进行测试。

例 5-29　处理经缓求值包装的异常

```
@Test
public void lazily_catching_other_peoples_exceptions() {
    P1<Either<Exception, Integer>> result = FjRomanNumeralParser.divideLazily(4, 2);
    assertEquals((long) 2, (long) result._1().right().value());
    P1<Either<Exception, Integer>> failure = FjRomanNumeralParser.divideLazily(4, 0);
    assertEquals("/ by zero", failure._1().left().value().getMessage());
}
```

在 Java 语言下建立 `Either` 模型并不轻松，因为 Java 语言本身没有这个概念，我们必须借用泛型和类来作为手工搭建 `Either` 模型的材料。Scala 语言内建了 `Either` 和其他类似用途的构造。Clojure 和 Groovy 语言没有内建类似 `Either` 的概念，那是因为它们都是动态类型

的语言，可以轻易生成所需类型的值。例如在 Clojure 下，我们不必特意建立一个双值的数据结构，返回 keyword 是更常见的办法，这是 Clojure 的一种字符串常量，可以用作标识符号。

5.5.3 Option类

Either 代表了一种使用方便、用途广泛的概念，除了用来实现双重返回值，还被用来构建一些通用的数据结构（如本章后面将讨论的基于 Either 的树结构）。在表示函数返回值这个用途上，有些语言和框架除了 Either 类之外，还存在另一个选项——Option。Option 类表述了异常处理中较为简化的一种场景，它的取值要么是 none，表示不存在有效值，要么是 some，表示成功返回。例 5-30 演示了 Functional Java 框架的 Option 实现。

例 5-30　Option 的用法

```
public static Option<Double> divide(double x, double y) {
    if (y == 0)
        return Option.none();
    return Option.some(x / y);
}
```

我们来测试一下上面经过 Option 类包装的返回值，请看例 5-31。

例 5-31　测试 Option 的行为

```
@Test
public void option_test_success() {
    Option result = FjRomanNumeralParser.divide(4.0, 2);
    assertEquals(2.0, (Double) result.some(), 0.1);
}

@Test
public void option_test_failure() {
    Option result = FjRomanNumeralParser.divide(4.0, 0);
    assertEquals(Option.none(), result);
}
```

我们从例 5-30 看到，Option 的取值为 none() 或 some() 其中之一，类似于 Either 的左值和右值，只不过专为表达“方法不一定返回有效结果”的意思而缩窄了定义。Option 类可以近似地看作 Either 类的一个子集；Option 一般只用来表示成功和失败两种情况，而 Either 可以容纳任意的内容。

5.5.4 Either树和模式匹配

我们还可以进一步拓展 Either 的用途，用它来构造一棵树形结构，并模仿 Scala 的模式匹配来为该结构实现遍历方法。

Either 利用 Java 的泛型支持，形成一种双重类型的数据结构，可以在其左值或右值上不同时地容纳两种类型的取值。

例如在前面的罗马数字解析示例中，我们让 Either 容纳了 Exception（左值）或者 Integer（右值）类型的取值，如图 5-1 所示。

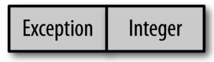


图 5-1：Either 抽象可容纳两种类型之一

该例中向 Either 装填内容的赋值操作发生在这一句：

```
Either<Exception, Integer> result = RomanNumeralParser.parseNumber("XLII");
```

接下来，我们准备在 Either 抽象的基础上构造一棵树形结构，这样做的好处要从模式匹配说起。

1. Scala的模式匹配

作为分发机制的模式匹配是 Scala 语言吸引人的特性之一。与其费笔墨描述，我们不如直接看一个例子，例 5-32 的函数将成绩数值转换成字母等级。

例 5-32　使用 Scala 模式匹配来实现成绩分等

```
val VALID_GRADES = Set("A", "B", "C", "D", "F")

def letterGrade(value: Any) : String = value match {
  case x:Int if (90 to 100).contains(x) => "A"
  case x:Int if (80 to 90).contains(x) => "B"
  case x:Int if (70 to 80).contains(x) => "C"
  case x:Int if (60 to 70).contains(x) => "D"
  case x:Int if (0 to 60).contains(x) => "F"
  case x:String if VALID_GRADES(x.toUpperCase) => x.toUpperCase
}
```

例 5-33 演示了这个成绩分等函数的使用效果。

例 5-33　测试成绩分等函数

```
printf("Amy的成绩为%d,获得%s等\n", 91, letterGrade(91))
printf("Bob的成绩为%d,获得%s等\n", 72, letterGrade(72))
printf("Sam缺席全部课程,成绩%d,获得%s等\n",
    44, letterGrade(44))
printf("Roy转学前已获%s等,记为%s等\n",
    "B", letterGrade("B"))
```

例 5-32 中，letterGrade 函数的整个函数体都是针对其参数不同取值的 match 块。我们设定了一系列的模式来防备每一种取值情况，除了参数的类型，这些模式还可以根据各种细致的条件来划分和筛选参数的取值。模式匹配的语法将每一种取值情况划分得清晰明了，比起笨拙的连串 if 语句高明多了。

模式匹配可以和 Scala 的 case 类联用，这种特殊性质的类可以帮助我们隐去 case 语句中冗长的条件判断，将之转移到 case 类的定义里。请看例 5-34 对不同颜色组合的匹配演示。

例 5-34　Scala 中针对 case 类的模式匹配

```
class Color(val red:Int, val green:Int, val blue:Int)

case class Red(r:Int) extends Color(r, 0, 0)
case class Green(g:Int) extends Color(0, g, 0)
case class Blue(b:Int) extends Color(0, 0, b)

def printColor(c:Color) = c match {
  case Red(v) => println("Red: " + v)
  case Green(v) => println("Green: " + v)
  case Blue(v) => println("Blue: " + v)
  case col:Color => {
    print("R: " + col.red + ", ")
    print("G: " + col.green + ", ")
    println("B: " + col.blue)
  }

  case null => println("invalid color")
}
```

例 5-34 首先创建了基类 Color，然后以 case 类的形式建立特化的版本来表示单色。为了在函数中判断传入的参数是哪一种颜色，我们使用了 match 来对所有可能的取值选项作模式匹配，最后还匹配处理了空值的情况。

Scala 的 case 类

面向对象系统，尤其是一些差异较大，而需要互相通信的系统之间，经常使用一些简单的类来作为数据的承载容器。由于类的这种用法十分盛行，Scala 索性专门设计了 case 类。case 类自动地附带了以下语法便利。

- 类名可以直接用作一个工厂方法。我们不必动用 new 关键字就可以构造一个新实例，如 `val bob = Person("Bob", 42)`。
- 经类参数列表传入的所有值都会自动被赋予 val 类型，也就是说它们都成了类中值不可变的内部字段。
- 编译器自动为 case 类生成合理的 `equals()`、`hashCode()` 和 `toString()` 默认实现。
- 编译器添加到类中的 `copy()` 方法可以通过返回新副本的形式，实现对原实例的字段修改。

我们从 case 类身上可以管窥 Java 平台上这些后继语言的作为和抱负，它们并不满足于修补 Java 语法上的毛病，而是已经对现代软件的运作有了更好的理解，并磨砺自身去适应趋势。语言就是这样随着时间演化的。

Java 不支持模式匹配，因此我们没办法写出像 Scala 那样清晰可读的分发代码。但如果让泛型和我们熟悉的数据结构联起手来，未必不能学到一点神韵。于是话题又回到了 `Either`。

2. `Either`树

建立一棵树的数据结构模型只需要三种抽象，如表 5-2 所示。

表5-2：构造一棵树需要的三种抽象

树的抽象	说明
empty	没有值的单元
leaf	放置了某种数据类型的值的单元
node	指向其他的 leaf 或 node

方便起见，我们直接使用 Functional Java 框架中的 `Either` 类。理论上，`Either` 抽象内用来放置数据的栏位可以扩展为任意的数量。例如声明 `Either<Empty, Either<Leaf, Node>>` 将构造出如图 5-2 所示的三值结构。

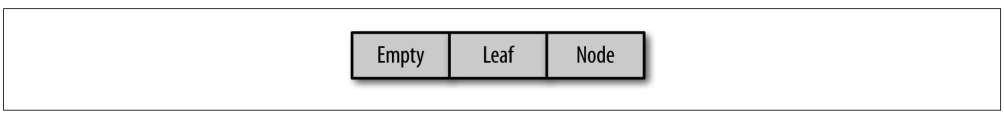


图 5-2：`Either<Empty, Either<Leaf, Node>>` 数据结构

借助这个把树的三种抽象组装到一起的 `Either` 结构，我们定义了例 5-35 的树。

例 5-35　在 `Either` 上建立起来的树结构

```
package com.nealford.ft.structuralpatternmatching;

import fj.data.Either;
import static fj.data.Either.left;
import static fj.data.Either.right;

public abstract class Tree {
    private Tree() {}

    public abstract Either<Empty, Either<Leaf, Node>> toEither();

    public static final class Empty extends Tree {
        public Either<Empty, Either<Leaf, Node>> toEither() {
            return left(this);
        }

        public Empty() {}
    }

    public static final class Leaf extends Tree {
        public final int n;

        @Override
```

```
        public Either<Empty, Either<Leaf, Node>> toEither() {
            return right(Either.<Leaf, Node>left(this));
        }

        public Leaf(int n) { this.n = n; }
    }

    public static final class Node extends Tree {
        public final Tree left;
        public final Tree right;

        public Either<Empty, Either<Leaf, Node>> toEither() {
            return right(Either.<Leaf, Node>right(this));
        }

        public Node(Tree left, Tree right) {
            this.left = left;
            this.right = right;
        }
    }
}
```

例 5-35 的 `Tree` 抽象类在其内部定义了三个 `final` 具体类：`Empty`、`Leaf`、`Node`。`Tree` 类内部使用如图 5-2 所示的三值 `Either` 来放置数据，它规定了 `Empty` 要放在最左边的位置，`Leaf` 放在中间，`Node` 放在最右边。每个内部类都实现了 `toEither()` 方法，保证本类的实例放在了三值 `Either` 中正确的栏位上。我们用来搭建树结构的构造单元，三值 `Either`，相当于传统计算机科学术语中"联合体"（union）的概念，虽然允许放入三种类型，但在任意时刻只会持有其中的一种。

我们有了树结构，还知道它的内部构造是 `<Empty, <Leaf, Node>>` 的样子，现在可以试着按照模式匹配的风格来实现树的遍历。

3. 以模式匹配的方式实现树的遍历

Scala 的模式匹配鼓励我们把不同的情况分开来考虑。Functional Java 框架中 `Either` 的 `left()` 和 `right()` 方法都实现了 `Iterable` 接口，我们模拟模式匹配的基本条件已经满足。例 5-36 按照模式匹配的风格实现了对树深度的检测。

例 5-36　模仿模式匹配的语法来获取树的深度

```
static public int depth(Tree t) {
    for (Empty e : t.toEither().left())
        return 0;
    for (Either<Leaf, Node> ln: t.toEither().right()) {
        for (Leaf leaf : ln.left())
            return 1;
        for (Node node : ln.right())
            return 1 + max(depth(node.left), depth(node.right));
    }
    throw new RuntimeException("Inexhaustible pattern match on tree");
}
```

例 5-36 的 `depth()` 方法是一个递归的深度查找函数。由于我们的树采用了特殊的单元结构（`<Empty, <Left, Node>>`），我们可以将每一个“栏位”都当成一个“case”来对待。如果单元是 empty，那么这根枝条深度为零。如果单元是 leaf，那么我们给树的深度累计一层。如果单元是 node，那么我们应该累计一层，并继续递归搜索单元的左子树和右子树。

我们还可以按照同样的模式匹配风格实现对树的递归搜索，如例 5-37 所示。

例 5-37　判断给定值在树中是否存在

```
static public boolean inTree(Tree t, int value) {
    for (Empty e : t.toEither().left())
        return false;
    for (Either<Leaf, Node> ln: t.toEither().right()) {
        for (Leaf leaf : ln.left())
            return value == leaf.n;
        for (Node node : ln.right())
            return inTree(node.left, value) | inTree(node.right, value);
    }
    return false;
}
```

例 5-37 的写法和例 5-36 一样，都是按照单元结构的“栏位”来返回相应结果。如果遇到 empty 单元，我们就返回 `false`，表示搜索失败了。如果遇到 leaf，我们检查单元中放置的值，如果匹配就返回 `true`。当遇到 node 单元的时候，我们继续递归搜索下游分支，用 `|`（非短路或运算符）合并左右子树的搜索结果。

我们可以做个单元测试来实际体验一下树的构造和搜索，如例 5-38 所示。

例 5-38　测试树的搜索

```
@Test
public void more_elaborate_searchp_test() {
    Tree t = new Node(new Node(new Node(new Node(
            new Node(new Leaf(4),new Empty()),
            new Leaf(12)), new Leaf(55)),
            new Empty()), new Leaf(4));
    assertTrue(inTree(t, 55));
    assertTrue(inTree(t, 4));
    assertTrue(inTree(t, 12));
    assertFalse(inTree(t, 42));
}
```

例 5-38 首先构造了一棵树，然后检测了几个值在树中是否存在。当树中任意叶子上的值与查找对象相同，`inTree()` 方法就返回 `true`。例 5-37 中用来合并节点左右子树查找结果的 `|` 运算符会使得 `true` 值沿着递归调用栈一路向上传播。

例 5-37 只检查了树中是否含有某元素，我们可以更进一步统计某元素在树中出现的次数，如例 5-39 所示。

例 5-39　计算元素在树中的重复次数

```
static public int occurrencesIn(Tree t, int value) {
    for (Empty e: t.toEither().left())
        return 0;
    for (Either<Leaf, Node> ln: t.toEither().right()) {
        for (Leaf leaf : ln.left())
            if (value == leaf.n) return 1;
        for (Node node : ln.right())
            return occurrencesIn(node.left, value)
                    + occurrencesIn(node.right, value);
    }
    return 0;
}
```

例 5-39 每找到一个符合的叶子结点就返回 1，经过递归累加就可以得出元素的重复次数。

例 5-40 在一棵复杂的树上检验了 depth()、inTree() 和 occurrencesIn() 函数。

例 5-40　在一棵复杂的树上获取深度、查找元素和计算元素的重复次数

```
@Test
public void multi_branch_tree_test() {
    Tree t = new Node(new Node(new Node(new Leaf(4),
            new Node(new Leaf(1), new Node(
                    new Node(new Node(new Node(
            new Node(new Node(new Leaf(10), new Leaf(0)),
                    new Leaf(22)), new Node(new Node(
                            new Node(new Leaf(4), new Empty()),
                            new Leaf(101)), new Leaf(555))),
                            new Leaf(201)), new Leaf(1000)),
                    new Leaf(4)))),
            new Leaf(12)), new Leaf(27));
    assertEquals(12, depth(t));
    assertTrue(inTree(t, 555));
    assertEquals(3, occurrencesIn(t, 4));
}
```

我们在树的内部结构上强加的规则，让元素的类型和遍历过程中可能遭遇的情况一一对应起来，因此我们可以把问题清晰地分成不同的情况来个别处理。虽然表达能力没有 Scala 的模式匹配那么强，但语法结构意外地相像。

第 6 章

模式与重用

如果我们的语言支持的编程范式以对象为本，我们就很容易不自觉地按照对象的术语来思考所有问题的答案。不过，大多数现代语言都是多范式的，支持对象、元对象、函数式，以及其他多样化的范式。学会使用不同的范式来处理不同的问题，是开发者进步路上需要越过的一道坎。

6.1 函数式语言中的设计模式

函数式世界中有一种代表性的意见认为，设计模式的概念本身在函数式编程中就站不住脚，函数式编程不需要这样的东西。假如按照比较狭窄的定义来解释模式的概念，这种观点是有道理的——但是把争辩的焦点从模式的用途转移到了“模式”的语义上。如果我们认可设计模式是“赋予了名字的、编目记录下来的常见问题的解决方案”，那么这个概念一点都不过时。只不过在不同的范式下，模式有可能呈现为截然不同的外在形象。因为函数式世界用来搭建程序的材料不一样了，所以解决问题的手法也不一样了，GoF 模式集中有一部分传统模式失去了存在的意义，但还有一部分模式，它们要解决的问题依然存在，只是解决的手段发生了很大的变化。

传统设计模式在函数式编程的世界中大致有三种归宿。

- 模式已被吸收成为语言的一部分。
- 模式中描述的解决办法在函数式范式下依然成立，但实现细节有所变化。
- 由于在新的语言或范式下获得了原本没有的能力，产生了新的解决方案（例如很多问题都可以用元编程干净利落地解决，但 Java 没有元编程能力可用）。

我们依次探讨以上三种情形。

6.2　函数级别的重用

复合（composition），作为一种重用机制，在函数式语言中主要表现为通过参数来传递作为第一等语言成分的函数，各种函数式编程库都频繁地运用了这种手法。与面向对象语言相比，函数式语言的重用发生于较粗的粒度级别上，着眼于提取一些共通的运作机制，并参数化地调整其行为。面向对象系统由一群互相发送通信消息（或者叫调用方法）的对象组成。图 6-1 描绘了这样的一个面向对象系统。

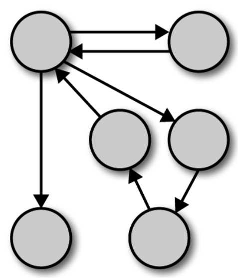


图 6-1：面向对象系统中的重用

如果我们从中发现了一小群有价值的类以及相应的消息，就可以将这部分类关系提取出来，加以重用，如图 6-2 所示。

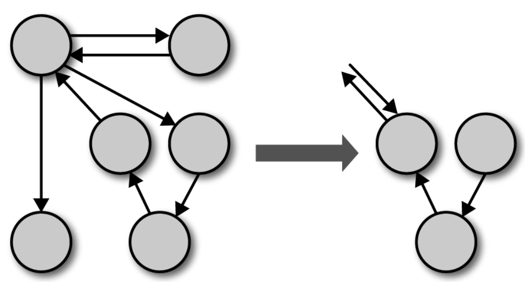


图 6-2：从类关系图中提取有用的部分

这方面集大成的《设计模式》毫无意外地成为了软件工程领域最深入人心的著作之一，它的工作就是汇总编目像图 6-2 那样提取出来的类关系。以模式为载体的重用随后遍地开花，编目和命名各方面模式的书籍纷纷涌现。这场设计模式运动极大地惠泽了软件开发世界，因为它为我们确立了可言说的名词术语和可参照的样例。不过从根本上说，以模式为载体的重用是细粒度的：一种解答方案（如 Flyweight 模式）与另一种解答方案（如 Memento 模式）之间，是井水不犯河水的“正交”关系。设计模式所解决的都是很专门的问题，模式的用处就在于我们经常能够为手头的问题找到对应的模式，但反过来，模式和问题之间

这种狭窄的对应关系又限制了它的适用面。

函数式程序员也喜欢重用代码，只是他们用了另外一套材料来搭建重用体系。函数式编程不追求复现结构之间经典的（耦合）关系，它以定义各类型“物件”之间“态射”（morphism）关系的数学分支——范畴论为基础，希望从代码中抽取另一种粗粒度的脉络而加以重用。大多数应用都离不开对列表元素的操作，函数式的重用机制就建立在列表的概念，以及可以连同执行上下文一起传递的代码块的概念之上。函数式语言依赖作为第一等语言成分的函数（第一等的函数允许出现在其他任何语言构造允许出现的位置上）去充当参数和返回值。图 6-3 描绘了这种思路。

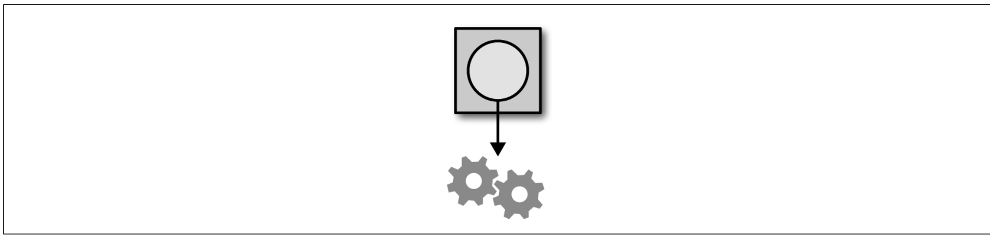


图 6-3：由可传递的代码和粗粒度机制构成的重用

图 6-3 的齿轮组代表一些宽泛地处理某种基本数据结构的抽象，方框代表可传递的代码，数据被封装在里面。

例 2-12 从命令式实现改写过来的函数式的列表筛选给了我们截然不同的体验，而它不过是遵照了各种函数式编程语言和库都习以为常的行文思路。我们从函数式语言把代码当作参数来传递（如传给例中的 `filter()` 方法）的能力中看到了一条不同于以往的代码重用途径。如果我们来自传统的奉设计模式为圭臬的面向对象世界，恐怕更习惯于构建各种类和方法来解决问题。

函数式语言不需要那么多辅助性的支撑结构和公式化的死板套语，它可以帮我们卸去一部分累赘，同时又实现与传统方式相同的设计概念。例如当语言拥有了闭包特性，就不需要 Command 模式了。从根本上说，设计模式的存在意义就是弥补语言功能上的弱点，例如不把“行为”用一个本身无甚意义的骨架类包装起来的话，就没办法像传递数值一样传递它。当然，Command 模式还有其他用途，例如实现操作*回退*（undo），不过它的主要功能是打开一条门路让代码块能够被传递到方法中去执行。

还有一种常用模式也在函数式语言中摆脱了公式化的的代码框框，它就是我们在第 3 章接触过的 Template Method 模式。

6.2.1 Template Method模式

函数被提升为第一等的语言成分，对 Template Method 模式的实现有简化的效果，因为可

以消除一些为了迂回语言限制而存在的结构。Template Method 模式在一个方法里面定义好算法的骨架，但留下一部分未实现的步骤，强迫子类按照规定好的算法结构来补全缺失的步骤定义。Template Method 模式的经典实现如例 6-1 的 Groovy 代码所示。

例 6-1　Template Method 模式的“标准”实现

```
package templates;

abstract class Customer {
    def plan

    def Customer() {
        plan = []
    }

    def abstract checkCredit()
    def abstract checkInventory()
    def abstract ship()

    def process() {
        checkCredit()
        checkInventory()
        ship()
    }
}
```

例 6-1 中 `process()` 方法依赖于 `checkCredit()`、`checkInventory()` 和 `ship()` 方法，这三个方法被设定为抽象方法，子类必须为它们提供具体实现。

由于作为第一等语言成分的函数可以取代任何语言结构，我们尝试用代码块来改写例 6-1 的实现，请看例 6-2。

例 6-2　第一等函数对 Template Method 模式的影响

```
package templates;

class CustomerBlocks {
    def plan, checkCredit, checkInventory, ship

    def CustomerBlocks() {
        plan = []
    }

    def process() {
        checkCredit()
        checkInventory()
        ship()
    }
}
```

原先需要特地按照规定格式来声明的算法步骤，在例 6-2 中成了类里面最普通不过的属性，可以像普通的属性一样赋值。这正是一个实现细节被语言特性吸收掉的例子。模式本身作

为针对某个问题的解决方案（将特定步骤推给下游环节去处理），还是一个有意义的讨论话题，只是模式的实现变得简单了。

前后两个实现并不等价。按照例 6-1 的 Template Method 模式“传统”实现，子类必须补全算法依赖的几个抽象方法的实现。虽然子类可以用空的方法体应付了事，但绝不可能全然无视这些空缺的方法。抽象方法的定义相当于一种特殊形式的文档，提醒子类将指定的方法纳入考虑。然而反过来，事先规定好全部的方法声明又有僵化之虞，未必适合需要更多灵活性的情况。例如我们也许希望任意指定 `Customer` 类处理时使用的方法序列。

语言对代码块等特性支持得越深入，对开发者的亲和力就越好。假设我们希望允许子类有时候跳过某些处理步骤。Groovy 有一个特殊的“受保护访问”运算符（`?.`），可以在调用方法之前先确认目标对象不是空值。于是我们可以写下如例 6-3 所示的 `process()` 方法定义。

例 6-3　为调用代码块增加防护

```
def process() {
  checkCredit?.call()
  checkInventory?.call()
  ship?.call()
}
```

例 6-3 免除了我们为 `checkCredit`、`checkInventory` 和 `ship` 属性赋值的义务，我们可以自由地决定要实现或留空哪一个函数。像 `?.` 运算符这样的语法糖衣是语言为开发者提供的帮助，让我们摆脱重复性的、难以梳理的死板代码，比如一大串的 `if`，代之以有表现力的简洁语句。`?.` 运算符本身谈不上有多么“函数式”，但它是把乏味工作推给运行时的一个好例子。

由语言直接提供的高阶函数特性可以让我们节约大量的八股代码，经典的 Command 模式和 Template Method 模式为我们做了最好的说明。

6.2.2　Strategy模式

Strategy 模式也是因为第一等函数而得到简化的一种常用模式。Strategy 模式定义一个算法族，并将每一种算法都在相同的接口下封装起来，令同一族的算法能够互换使用。这样做的好处是算法的变化不影响使用方，也不受使用方的影响。第一等函数让建立和操纵各种策略的工作变得十分简单。

例 6-4 给出了 Strategy 模式的一个传统实现，我们用它为两个数字的积的计算问题安排算法。

例 6-4　用 Strategy 模式来处理两个数字的积的计算问题

```
interface Calc {
  def product(n, m)
}
```

```
class CalcMult implements Calc {
  def product(n, m) { n * m }
}

class CalcAdds implements Calc {
  def product(n, m) {
    def result = 0
    n.times {
      result += m
    }
    result
  }
}
```

例 6-4 为计算两个数字的积定义了统一的接口。我们在接口下实现了两个具体类（也就是两种策略），一个直接用乘法计算，另一个用累加的方法。我们来测试一下效果，如例 6-5 所示。

例 6-5　测试积的计算策略

```
class StrategyTest {
  def listOfStrategies = [new CalcMult(), new CalcAdds()]

  @Test
  public void product_verifier() {
    listOfStrategies.each { s ->
      assertEquals(10, s.product(5, 2))
  }
}
```

不出所料，例 6-5 的两种策略都返回了相同的结果。然而例 6-4 公式化的累赘成分太多了，假如我们正确发挥 Groovy 代码块作为第一等函数的能力，应该可以做得更好。请看例 6-6 对幂的不同计算策略的表达。

例 6-6　简洁地表达和测试幂的不同计算策略

```
@Test
public void exp_verifier() {
  def listOfExp = [
      {i, j -> Math.pow(i, j)},
      {i, j ->
        def result = i
        (j-1).times { result *= i }
        result
      }]

  listOfExp.each { e ->
    assertEquals(32, e(2, 5))
    assertEquals(100, e(10, 2))
    assertEquals(1000, e(10, 3))
  }
}
```

例 6-6 的两种幂计算策略都是以 Groovy 代码块的形式就地定义的，我们为表述上的便利而牺牲了规范的形式。传统手法规定了每种策略的名称和结构，在某些情况下更可取。但例 6-6 的好处是我们可以随时给代码增加更严格的防护，而想要突破传统方式设下的框框就不那么容易了。这几个例子不完全是函数式编程和设计模式的对比，它们更多地表现了动态类型和静态类型思维的对立。

6.2.3 Flyweight模式和记忆

Flyweight 模式是一种在大量的细粒度对象引用之间共享数据的优化技巧。我们维护一个对象池，然后引用池中的对象来构成需要的视图。

Flyweight 模式使用了“标准品”对象的概念——可以代表所有其他同型对象的一个典型的对象实例。例如我们可以用某种消费产品的一个“标准品”来代表同型的所有产品。

同理，我们在程序中没必要为每个用户都创建各自的产品对象列表，相反可以只创建一份标准品的列表，让用户引用列表中的对象来表示自己持有的产品。例 6-7 建立了若干计算机品类的模型。

例 6-7　建立计算机品类模型的一些简单类

```
class Computer {
  def type
  def cpu
  def memory
  def hardDrive
  def cd
}

class Desktop extends Computer {
  def driveBays
  def fanWattage
  def videoCard
}

class Laptop extends Computer {
  def usbPorts
  def dockingBay
}

class AssignedComputer {
  def computerType
  def userId

  public AssignedComputer(computerType, userId) {
    this.computerType = computerType
    this.userId = userId
  }
}
```

按照这些类的设计，假如所有计算机都是一样的配置，我们为每个用户都创建一个 `Computer` 新实例就很不经济了。`AssignedComputer` 对象的作用是把计算机关联到用户。

联用 Factory 和 Flyweight 模式是改善上述设计的常用方法。我们可以设计一个生产计算机标准品的单例工厂，如例 6-8 所示。

例 6-8　生产 flyweight 计算机实例的单例工厂

```
class CompFactory {
  def types = [:]
  static def instance;

  private ComputerFactory() {
    def laptop = new Laptop()
    def tower = new Desktop()
    types.put("MacBookPro6_2", laptop)
    types.put("SunTower", tower)
  }

  static def getInstance() {
    if (instance == null)
      instance = new CompFactory()
    instance
  }

  def ofType(computer) {
    types[computer]
  }
}
```

`ComputerFactory` 类建立了一个缓存来放置可能的计算机品类，外界通过它的 `ofType()` 方法来索取需要的实例。例 6-8 是非常传统的单例工厂写法，如果我们用 Java 语言来实现的话，差不多就是这个样子。

不过，Singleton 模式本身，也是模式被运行时吸收掉的典型案例。我们可以利用 Groovy 提供的 `@Singleton` 标注来简化 `ComputerFactory` 的实现，如例 6-9 所示。

例 6-9　简化的单例工厂

```
@Singleton class ComputerFactory {
  def types = [:]

  private ComputerFactory() {
    def laptop = new Laptop()
    def tower = new Desktop()
    types.put("MacBookPro6_2", laptop)
    types.put("SunTower", tower)
  }

  def ofType(computer) {
    types[computer]
  }
}
```

我们来测试一下工厂返回的标准品实例，请看例 6-10。

例 6-10　证明工厂返回的是标准品

```
@Test
public void comp_factory() {
  def bob = new AssignedComputer(
    CompFactory.instance.ofType("MacBookPro6_2"), "Bob")
  def steve = new AssignedComputer(
    CompFactory.instance.ofType("MacBookPro6_2"), "Steve")
  assertTrue(bob.computerType == steve.computerType)
}
```

把不同实例共用的信息保存起来是个好主意，我们希望把它也带到函数式编程中去，只是具体的做法会有很大不同。我们将看到一个保留模式的语义，但改变（最好是简化）其实现的例子。

我们在第 4 章讨论过函数的“记忆”，被记忆的函数允许运行时缓存其结果。请看例 6-11 定义的函数及其记忆版本。

例 6-11　把共享内容记忆起来

```
def computerOf = {type ->
  def of = [MacBookPro6_2: new Laptop(), SunTower: new Desktop()]
  return of[type]
}

def computerOfType = computerOf.memoize()
```

例 6-11 在 `computerOf` 函数内定义了标准品。我们只需要在 `computerOf` 函数上调用 `memoize()` 方法，就可以得到相应的带记忆能力的函数实例。

例 6-12 分别测试了用单例工厂方式实现和用记忆方式实现的两组用例。

例 6-12　两种实现方式的比较

```
@Test
public void flyweight_computers() {
  def bob = new AssignedComputer(
    ComputerFactory.instance.ofType("MacBookPro6_2"), "Bob")
  def steve = new AssignedComputer(
    ComputerFactory.instance.ofType("MacBookPro6_2"), "Steve")
  assertTrue(bob.computerType == steve.computerType)

  def sally = new AssignedComputer(
    computerOfType("MacBookPro6_2"), "Sally")
  def betty = new AssignedComputer(
    computerOfType("MacBookPro6_2"), "Betty")
  assertTrue sally.computerType == betty.computerType
}
```

两组测试的结果相同，但它们的实现细节差别巨大。遵照“传统的”设计模式，我们要创

建一个充当工厂的新类，并实现两个方法。在函数式的版本里，我们实现了一个方法，然后得到它的带记忆实例。当运行时接管像缓存这样的实现细节，就意味着减少了手工编写犯错的可能性。我们最后得到一个保留了 Flyweight 模式的语义，而且非常简单的函数式实现。

6.2.4 Factory模式和柯里化

在设计模式的语境下，柯里化相当于产出函数的工厂。第一等函数（或高阶函数）是函数式编程语言共同的特性，我们可以用函数来充当其他任何的语言成分。因此我们可以很容易地设立一个根据条件来返回其他函数的函数，也就是函数工厂。我们可以看一个例子，假设有一个用于两数相加的普通函数，经过柯里化加工，我们可以制造出一个总是在其参数上加一的递增函数，如例 6-13 的 Groovy 代码所示。

例 6-13　作为函数工厂的柯里化

```
def adder = { x, y -> x + y}
def incrementer = adder.curry(1)

println "7 的递增：${incrementer(7)}"
```

例 6-13 在 `adder` 上通过柯里化把第一个参数固定为 `1`，这就是我们的函数工厂，它会为我们产出一个单参数的函数。

我们有必要回顾一个第 3 章演示过的例子，请看例 6-14 的 Scala 递归筛选示例。

例 6-14　递归式的筛选函数，Scala 实现

```
object CurryTest extends App {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def dividesBy(n: Int)(x: Int) = ((x % n) == 0) // ❶

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, dividesBy(2))) // ❷
  println(filter(nums, dividesBy(3)))
}
```

❶ 定义时已经指明函数将被柯里化使用。

❷ `filter` 要求传入一个集合（`nums`）和一个单参数的函数（柯里化之后 `dividesBy()` 函数就变成单参数了）。

在拘谨的模式眼光看来，例 6-14“不经意”地柯里化了 `dividesBy()` 方法，有些不可思议。`dividesBy()` 本来接受两个参数，并根据第二个参数能否被第一个参数整除来返回 `true` 或

false。然而在它参与到 filter() 操作的时候，我们只为它提供了一个参数，以柯里化的方式来制造一个单参数的函数去充当 filter() 方法的谓词。

模式在函数式编程下有三种归宿，这个例子同时反映了其中的两种。首先，柯里化已经内建在语言或运行时里面，函数式工厂的概念天然存在，不需要我们添加额外的结构。其次，柯里化带给我们一种全新的实现途径。恪守 Java 的程序员怎么都想不到会有例 6-14 那样的柯里化解法，他们不曾拥有真正的可传递的代码，对于在通用函数上构造专用函数的思路更为陌生。甚至很可能，大多数习惯于命令式编程的开发者根本没想过在这种地方使用设计模式，毕竟在通用版本的基础上构造一个特化的 dividesBy() 方法，看起来只是一个小问题，而设计模式是为了更大型的问题准备的，因为主要依赖结构来解决问题的设计模式有着很重的实现负担。柯里化交给我们的答案如此轻巧，都不值得像模式那样专门为解答方案起一个名字，因为柯里化只是在做它的本职工作而已。

柯里化可以把通用的函数改造成专用的函数。

6.3 结构化重用和函数式重用的对比

第 1 章有这么一段引言：

> 面向对象编程通过封装不确定因素来使代码能被人理解；函数式编程通过尽量减少不确定因素来使代码能被人理解。
>
> ——Michael Feathers

每天在同一种抽象的包围下工作，我们的头脑会逐渐被它浸染，我们处理问题的方式也会被潜移默化。这一节我们将尝试通过重构来实现代码重用，并剖析抽象方式对思维角度的影响。

简化状态的封装和使用，是面向对象的目标之一。自然地，状态就成了面向对象的抽象用来解决问题的常规武器，维系状态所需的众多类和交互也因此被派生出来——这些正是 Michael Feathers 所说的“不确定因素”。

函数式编程不喜欢把结构耦合在一起，它依靠零件之间的复合来组织抽象，以达到减少不确定因素的目的。开发者如果只有面向对象语言的经验，会不容易理解这里面的微妙差别。

以结构为载体的代码重用

命令式的、（尤其是）面向对象的编程风格，使用结构和消息作为建筑材料。如果一段面向对象的代码值得重用，那么我们会把它提取到另一个类中，然后通过继承来访问它。

为了演示这种基于结构的代码重用和它的隐含后果，我准备了两个例子。第一个是我们熟悉的完美数分类例子，它可以展示代码的结构和风格。

第二个例子是通过一个正整数的约数来判断它是否为素数（即大于 1，且除了 1 和它本身之外，没有其他约数的整数）。由于两个例子都需要求出一个数的约数，我们可以在下一步考虑通过重构来复用与约数相关的部分，并在重构的过程中展示不同风格的代码重用。

例 6-15 是以命令式风格实现的完美数分类程序。

例 6-15　命令式的完美数分类实现

```
public class ClassifierAlpha {
    private int number;

    public ClassifierAlpha(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> factors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    static public int sum(Set<Integer> factors) {
        Iterator it = factors.iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
    }

    public boolean isPerfect() {
        return sum(factors()) - number == number;
    }

    public boolean isAbundant() {
        return sum(factors()) - number > number;
```

```
    }

    public boolean isDeficient() {
        return sum(factors()) - number < number;
    }
}
```

我们在第 2 章已经分析过上述实现的演变和改造，不必再次重复，这里单纯是借它来解说代码重用。例 6-16 同为命令式风格的素数判定程序也是我们的解说材料。

例 6-16　命令式的素数判定实现

```
public class PrimeAlpha {
    private int number;

    public PrimeAlpha(int number) {
        this.number = number;
    }

    public boolean isPrime() {
        Set<Integer> primeSet = new HashSet<Integer>() {{
            add(1); add(number);}};
        return number > 1 &&
                factors().equals(primeSet);
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> factors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }
}
```

例 6-16 有几个地方值得注意。第一个地方是 `isPrime()` 方法中有点奇特的初始化代码。我们使用的语法结构叫作*实例初始化块*（instance initializer），是 Java 的一种构造技巧。这个代码块放在类里面，但又不从属于任何方法，它是 Java 在构造器之外提供的另一种控制实例创建的手段。

例 6-16 的 `isFactor()` 方法和 `factors()` 方法也值得我们注意。它们和 `ClassifierAlpha` 类（见例 6-15）中的承担相同职责的同名函数一模一样。我们在独立实现了两套解决方案之后才发现，原来它们的功能是一样的。

1. 用重构来消灭重复

功能重复的问题可以用重构来解决，我们把重复的部分单独提取到新的 `Factors` 类，如例 6-17 所示。

例 6-17　经过提取、重构的共通代码

```
public class FactorsBeta {
    protected int number;

    public FactorsBeta(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential_factor) {
        return number % potential_factor == 0;
    }

    public Set<Integer> getFactors() {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }
}
```

我们可以直接使用 IDE 的*提取超类*（extract superclass）功能来得到例 6-17 的结果。由于被提取的两个方法都使用了成员变量 `number`，因此 `number` 也一起搬到了超类。执行重构的时候，IDE 会询问我们怎样处理 `number` 的访问（生成读、写方法，设定 `protected` 前缀等）。例中选择将 `number` 设为 `protected` *作用域*，因此 IDE 把它设为新类中的一个字段，并生成赋值用的构造器。

经过隔离和移除重复代码的改造，完美数分类和素数判定两个例子都大大地简化了。例 6-18 是重构之后的完美数分类程序。

例 6-18　重构之后简化了的完美数分类程序

```
public class ClassifierBeta extends FactorsBeta {

    public ClassifierBeta(int number) {
        super(number);
    }

    public int sum() {
        Iterator it = getFactors().iterator();
        int sum = 0;
        while (it.hasNext())
            sum += (Integer) it.next();
        return sum;
```

```java
    }

    public boolean isPerfect() {
        return sum() - number == number;
    }

    public boolean isAbundant() {
        return sum() - number > number;
    }

    public boolean isDeficient() {
        return sum() - number < number;
    }
}
```

例 6-19 是重构之后的素数判定程序。

例 6-19　重构之后简化了的素数判定程序

```java
public class PrimeBeta extends FactorsBeta {
    public PrimeBeta(int number) {
        super(number);
    }

    public boolean isPrime() {
        Set<Integer> primeSet = new HashSet<Integer>() {{
            add(1); add(number);}};
        return getFactors().equals(primeSet);
    }
}
```

无论我们重构时怎样安排 `number` 字段的访问选项，在决策的时候都不能只考虑当前的类，而要把关系网里所有的类都考虑进来。很多时候这种思考是有益的，因为可以帮助我们划定问题的边界。坏处是父类中的修改会影响下游的类。

这是一种通过耦合来实现的代码重用：两个代码单元（`ClassifierBeta` 类和 `PrimeBeta` 类）因为共享了父类的 `number` 字段和 `getFactors()` 方法，而被绑在了一起。语言本身的耦合规则促成了这种用法。面向对象规定了耦合式的交互风格（例如规定我们通过继承来获得对成员变量的访问权），对于各种事物如何耦合我们有一套预定的规则——这是好事，因为可以一致地推理各种情况下的行为。继承是很有用的特性，只是面向对象语言滥用了继承，有时候别的一些抽象更符合需要。

2. 以复合的方式实现的重用

第 2 章曾经在 Java 下实现过一个函数式版本的完美数分类程序，如例 6-20 所示。

例 6-20　稍具函数式特征的完美数分类实现

```java
import java.util.Collection;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;
```

```java
public class NumberClassifier {

    public static boolean isFactor(final int candidate, final int number) {   ❶
        return number % candidate == 0;
    }

    public static Set<Integer> factors(final int number) {                    ❷
        Set<Integer> factors = new HashSet<>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < number; i++)
            if (isFactor(i, number))
                factors.add(i);
        return factors;
    }

    public static int aliquotSum(final Collection<Integer> factors) {         ❸
        int sum = 0;
        int targetNumber = Collections.max(factors);
        for (int n : factors) {
            sum += n;
        }
        return sum - targetNumber;
    }

    public static boolean isPerfect(final int number) {                       ❹
        return aliquotSum(factors(number)) == number;
    }

    public static boolean isAbundant(final int number) {
        return aliquotSum(factors(number)) > number;
    }

    public static boolean isDeficient(final int number) {
        return aliquotSum(factors(number)) < number;
    }
}
```

❶ 各方法都必须加上 `number` 参数，因为没有可以存放它的内部状态。

❷ 所有方法都带 `public static` 修饰，因为它们都是*纯函数*，并因此可以一般地应用于完美数分类之外的领域。

❸ 注意例中对参数类型的选取，尽可能宽泛的参数类型可以增加函数重用的机会。

❹ 无缓存，在重复执行分类操作的情况下效率低下。

例 6-16 也可以改写成（使用纯函数，无共享状态的）函数式版本，其中的 `isPrime()` 方法如例 6-21 所示。其他方法都与例 6-20 中同名方法的实现完全一致。

例 6-21　函数式版本的素数判定程序

```java
public class FPrime {

    public static boolean isPrime(int number) {
```

```
        Set<Integer> factors = Factors.of(number);
        return number > 1 &&
                factors.size() == 2 &&
                factors.contains(1) &&
                factors.contains(number);
    }
}
```

我们像前面命令式版本那样，把重复的代码提取出来，放进单独的 Factors 类。再把 factors() 方法改名为 of()，以提高代码的可读性。结果如例 6-22 所示。

例 6-22　经过函数式重构的 Factors 类

```
public class Factors {
    static public boolean isFactor(int number, int potential_factor) {
        return number % potential_factor == 0;
    }

    static public Set<Integer> of(int number) {
        HashSet<Integer> factors = new HashSet<>();
        for (int i = 1; i <= sqrt(number); i++)
            if (isFactor(number, i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }
}
```

因为函数式版本的实现通过参数来传递所有的状态，我们提取出来的重用代码也就不包含任何共享的状态。我们接下来就可以在 Factors 类的基础上重构完美数分类和素数查找程序了。例 6-23 是完美数分类程序的重构结果。

例 6-23　经过重构的完美数分类程序

```
public class FClassifier {

    public static int sumOfFactors(int number) {
        Iterator<Integer> it = Factors.of(number).iterator();
        int sum = 0;
        while (it.hasNext())
            sum += it.next();
        return sum;
    }

    public static boolean isPerfect(int number) {
        return sumOfFactors(number) - number == number;
    }

    public static boolean isAbundant(int number) {
        return sumOfFactors(number) - number > number;
    }

    public static boolean isDeficient(int number) {
```

```
            return sumOfFactors(number) - number < number;
        }
    }
```

我们并没有借助任何特殊的类或语言来提高最后这一版实现的函数式程度。我们通过复合（composition）而不是耦合（coupling）来达到代码重用的目的。例 6-23 使用了 `Factors` 类，但使用关系被限制在单个方法的范围。

耦合与复合的差别是微妙的，但也是重要的。这一节的示例算不上复杂，所以耦合关系不至于妨碍我们看清代码结构的骨架。可是万一需要重构大量代码的时候，我们就会发现处处都会遇到耦合，因为耦合是面向对象语言非常基本的重用机制。盘根错节、难以理解的耦合关系已经妨害了面向对象语言下的代码重用，只在一些非常成熟的技术领域，如对象关系映射、图形界面组件库等少数方面，才取得了令人满意的效果。而在那些面向对象特征不那么明显的领域（例如各种业务应用），高水平的重用只是美好的愿望。

我们对命令式版本的重构本应该做得更好，假如我们能及时注意到 IDE 提议向父类添加的 `protected` 成员，制造了父类和子类之间的耦合；假如我们注意到复合是更合适的手段。像函数式程序员一样思考，意味着要用新的视角去看待编程中的一切方面。代码重用是不同编程范式的共同追求，但函数式程序员在这个问题上有着不同于命令式抽象的解决途径。

本节开头列举过设计模式在函数式编程下的几种归宿。第一，被语言或运行时吸收掉的模式，我们举了 Factory、Strategy、Singleton 和 Template Method 模式的例子。第二，模式保留了原来的语义，但实现发生了彻底的变化，我们对比了 Flyweight 模式基于类的实现和基于记忆特性的实现。第三，函数式的语言和运行时的语言特性不同以往，它们解决问题的方式也不同以往。

第 7 章

现实应用

到目前为止，我们演示的例子都比较抽象，因为主要目的是说明函数式编程迥异的思维模式。不过当学习到一定程度之后，我们肯定要回到现实中来检验学到的理论。

我们就用这一章来探访函数式思维在现实世界中的身影。我们从 Java 8 开始，然后讨论各种函数式的架构和 Web 框架。

7.1 Java 8

我们已经在本书前面的一些例子里演示过 Java 8 的 lambda 语法。设计 Java 8 的工程师们很聪明，他们没有生硬地在语言中安插高阶函数，而是巧妙地让旧的接口也享受到函数式新特性的好处。

我们为第 2 章的“公司业务处理”例子写过一个 Java 8 的实现版本，如例 7-1 所示。

例 7-1 公司业务处理流程的 Java 8 实现

```
public String cleanNames(List<String> names) {
    if (names == null) return "";
    return names
            .stream()
            .filter(name -> name.length() > 1)
            .map(name -> capitalize(name))
            .collect(Collectors.joining(","));
}

private String capitalize(String e) {
    return e.substring(0, 1).toUpperCase() + e.substring(1, e.length());
}
```

Java 8 的 stream 上可以串联组合一连串的函数，直到我们调用一个产生输出的函数（称为终结操作），如 `collect()`、`forEach()` 来终结这种串联。例 7-1 的 `filter()` 方法就是我们反复用来举例的筛选函数。`filter()` 方法的参数是一个返回 Boolean 值的高阶函数，它用作筛选的指示条件：返回 `true` 表示应放入筛选结果，返回 `false` 则表示应从结果集合中剔除。

Java 8 除了增加函数式特性，还增加了一些配合使用的语法糖衣。例如例 7-1 中的 `filter(name -> name.length() > 1)`，完整写法其实应该是 `filter((name) -> name.length() > 1)`。因为被传递的 lambda 块只有一个参数，所以允许省略意义不大的括号。stream 操作有时候还可以把返回类型"隐藏"起来。

`filter()` 方法要求传入的参数类型 `Predicate<T>`，其实就是一个返回 Boolean 值的函数。我们愿意的话，也可以显式地创建出谓词的实例，如例 7-2 所示。

例 7-2　手工创建一个谓词实例

```
Predicate<String> p = (name) -> name.startsWith("Mr");
List<String> l = List.of("Mr Rogers", "Ms Robinson", "Mr Ed");
l.stream().filter(p).forEach(i -> System.out.println(i));
```

例 7-2 通过赋值来创建谓词的实例，被赋值给谓词变量的是作为筛选条件的 lambda 块。最后在第三行调用 `filter()` 方法的时候，把谓词实例当作参数传了进去。

例 7-1 在筛选之后，又调用了 `map()` 方法来对集合中的每一个元素施用 `capitalize()`。最后调用的 `collect()` 方法是一个"终结操作"——从 stream 中产出结果值的操作。`collect()` 执行的是我们熟悉的化约操作：将集合中的元素结合成（一般）数目较少的结果值，甚至单一值（如求和操作）。Java 8 有专门的 `reduce()` 方法，但这里用 `collect()` 更合适，因为它处理值可变的容器（如 `StringBuilder`）的效率更高。

Java 要想让现有的类和集合适应 map、reduce 等函数式构造，那么集合的更新效率是必须处理好的一个难题。否则，如果像 `ArrayList` 那么典型的 Java 集合都无法执行化约操作，reduce 的意义就大打折扣了。Scala 和 Clojure 的集合库一般默认使用值不可变的类型，便于运行时生成高效率的操作。Java 8 没办法强迫开发者放弃原来的集合，而原来的集合大多是值可变的。在这种情况下，Java 8 提供了可以在 `ArrayList` 和 `StringBuilder` 等值可变的集合上执行化约操作的方法，这些方法不再把结果输出到另外的副本，而是直接在集合上更新元素。例 7-1 用 `reduce()` 方法也能求出同样的结果，但例中的返回集合用 `collect()` 效率更高。掌握这种细微的差别，是我们为了在现有语言上获得函数式的强大能力而付出的一点额外成本。

7.1.1　函数式接口

含有单一方法的接口是 Java 的一种习惯用法，称为 SAM（Single Abstract Method，单抽

象方法）接口，`Runnable` 和 `Callable` 接口都是有代表性的例子。很多时候，SAM 接口主要被当成一种将代码传递到异地执行的机制来使用。现在 Java 8 有了 lambda 块这种更好的传递代码的媒介。而我们通过一种叫作“函数式接口”的巧妙机制，可以让 lambda 和 SAM 携起手来。函数式接口是对旧有 SAM 接口的增强，它允许我们用 lambda 块取代传统的匿名类来就地实例化一个接口。例如 `Runnable` 接口就被加上了 `@FunctionalInterface` 标注，于是，编译器知道并验证 `Runnable` 确实是一个接口（而非 `class` 或 `enum`），并且符合函数式接口的要求。

我们可以在例子中感受一下取代 `Runnable` 匿名内部类的 Java 8 新语法，下面的代码通过传递 lambda 块来创建了新的线程：

```
new Thread(() -> System.out.println("Inside thread")).start();
```

函数式接口可以和 lambda 块默契地配合，在很多场合发挥效用。函数式接口的创意非常优秀，因为它们很好地照顾了 Java 长久以来的习惯用法。

Java 8 还允许我们在接口上声明*默认方法*。“默认方法”是一些在接口类型中声明的，以 `default` 关键字标记的，非抽象、非静态的 `public` 方法（且带有方法体定义）。默认方法会被自动地添加到实现了接口的类中，这就为我们提供了一条在类上“装饰”默认功能的方便途径。善用这种特性的例子，如 `Comparator` 接口，其默认方法多达十余个。我们用 lambda 块实现的比较器，可以极其简单地衍生出另一个反向的比较器，如例 7-3 所示。

例 7-3　`Comparator` 接口中的默认方法

```
List<Integer> n = List.of(1, 4, 45, 12, 5, 6, 9, 101);
Comparator<Integer> c1 = (x, y) -> x - y;
Comparator<Integer> c2 = c1.reversed();
System.out.println("Smallest = " + n.stream().min(c1).get());
System.out.println("Largest = " + n.stream().min(c2).get());
```

例 7-3 给 lambda 块套上一层外皮，创建出一个 `Comparator` 实例。然后，我们只需要调用一下现成的默认方法 `reversed()`，就得到一个颠倒了方向的比较器实例。这种在接口上附着默认方法的能力，可以认为是对“mixin”特性的一种模仿，也是对 Java 语言的有益补充。

我们可以在好几种语言中找到 mixin 的概念。它最早源自 Flavors 语言（http://dwz.cn/wiki-flavors-lang），是受到开发地点附近一家冰淇淋店的启发而产生的。那家店可以按照顾客的喜好，在原味的冰淇淋上“混入”任意的浇头（糖酥、彩糖珠、果仁碎，等等）。

有些早期的面向对象语言规定，类的属性和方法都集中定义在同一处，由单个的代码块来构成完整的类定义。而另外一些语言则允许开发者先定义属性，以后再择机完成方法的定义，然后“混入”类中。随着面向对象语言的演化，mixin 特性在现代语言中的实现原理和细节也发生了变化。

Ruby、Groovy 等类似语言也允许通过 mixin 的形式，在既有的类层次上增补功能。这些语言中的 mixin 是介于接口和父类之间的一种结构。它和接口一样都是类型，都可以执行 `instanceof` 检查，也都遵循一样的扩展规则。同一个类上可以混入不限数目的 mixin。但有一点和接口不一样，mixin 除了规定方法的签名，还可以实现该签名对应的行为。Java 8 的默认方法向 Java 语言引入了 mixin 机制，从此 JDK 不需要再单纯为了给静态方法找个归属，而生硬地设置 `Arrays`、`Collections` 这样的类了 。

7.1.2 Optional类型

例 7-3 最后两行的调用都以 `get()` 结尾，这里面有另一项 Java 8 特性的身影。Java 8 的 `min()` 等内建方法都不直接返回结果值，而是返回一个 `Optional` 结构。我们在第 5 章讨论过类似的行为。`Optional` 防止方法的返回结果出现无法区分表示错误的 `null` 和作为有效结果的 `null` 的情况。Java 8 还提供了 `ifPresent()` 方法，可以用在终结操作的位置上，设定在仅当存在有效结果时执行的一个代码块。例如下面的例子仅在有返回值的时候打印出结果：

```
n.stream()
    .min((x, y) -> x - y)
    .ifPresent(z -> System.out.println("smallest is " + z));
```

如果我们还想处理其他情况，可以求助于功能类似的 `orElse()` 方法。Java 8 的 `Comparator` 接口是默认方法非常耀眼的例子，我们从它身上可以深切地体会到默认方法的威力。

7.1.3 Java 8的stream

很多函数式的语言、框架（如 Functional Java，http://functionaljava.org/）都纳入了 stream 抽象，它们的实现各有微妙的区别。Java 8 也加入了这个行列，提供了一批有代表性的 stream 相关特性。

Java 8 的 stream 抽象为众多高级的函数式特性奠定了基础。stream 在很多方面的行为都与集合相似，但有一些关键的区别。

- stream 不存储值，只担当从输入源引出的管道角色，一直连接到终结操作上产生输出。
- stream 从设计上就偏向函数式风格，避免与状态发生关联。例如 `filter()` 操作在返回筛选结果的 stream 时，并不会改动底下的集合。
- stream 上的操作尽可能做到缓求值（详见第 4 章）。
- stream 可以没有边界（无限长）。例如我们可以构造一个返回所有数字的 stream，然后用 `limit()`、`findFirst()` 等方法来取得其一部分子集。
- stream 像 `Iterator` 的实例一样，也是消耗品，用过之后必须重新生成新的 stream 才能再次操作。

stream 的操作分为中间操作和终结操作。中间操作一律返回新的 stream，并且总是缓求值的。例如在 stream 上调用 `filter()` 操作，并不会真的在 stream 上进行筛选，而是产生一个新的 stream，让它只为终结操作的遍历过程提供满足筛选条件的值。终结操作遍历 stream，产生结果值和副作用（如果我们让函数产生副作用的话；虽然不鼓励，但是允许）。

7.2 函数式的基础设施

本书的大多数例子只在很小的规模上表现了函数式编程的优势，诸如用闭包来替代 Command 设计模式，记忆特性的使用技巧等。那么如果我们把函数式的思维运用到更大规模的事物，例如开发者们每天都要打交道的数据库、软件架构上，又会怎么样呢？

从匿名内部类到 lambda 块的转换并不困难，因为 Java 语言的设计者搭建了很好的解决机制，我们可以轻松地用函数式的构造一点一点把程序替换掉。可是，如果我们想对软件架构以及处理数据的基本方式实施类似的渐进转换，将会困难得多。我们就用下面的几个小节来谈谈函数式编程在实践中的影响。

7.2.1 架构

函数式的架构从根本上贯彻“值不可变”的思路，最大化地发挥其优点。学会从值不可变的角度去思考，是我们掌握函数式程序员的思维方法的一条重要门径。虽然在 Java 语言下构造值不可变的对象会带来一点前期的复杂性，但比起值不可变抽象对后续工作的简化作用，这点辛苦是完全值得的。

值不可变的类可以摆脱 Java 开发中一大批典型的负累。在函数式的思维下，我们会意识到，测试是为了确认代码中成功地制造了我们需要的变化。换言之，测试的真正目的是对可变事物的检验——可变的事物越多，就需要越多的测试来保证其正确性。

> 可变的状态与测试数量有直接的关联：可变的状态越多，要求的测试也越多。

如果我们严格限制可变性，只允许在规定的地方制造变化，那么就大大缩小了可能出错的位置范围，需要测试的地方也随之减少。对于值不可变的类来说，由于变化只发生在构造期间，它的单元测试也就变得十分轻松了。并且我们不再需要设置复制构造器（copy constructor），也不再需要折腾 `clone()` 方法棘手的实现细节。值不可变的对象很适合充当 map 和 set 数据结构中的键；Java 不允许字典型集合中的键在它被集合引用期间发生取值的变化，值不可变的对象完全符合这项要求。

值不可变的对象天生就是线程安全的，完全不会发生同步方面的问题。它们还绝对不会由

于异常而处于不明确的、预料之外的状态。因为所有的初始化过程都发生在对象的构造期间，而 Java 的对象构造具有原子性，也就是说，异常只会发生在我们得到对象实例之前。Joshua Bloch 称这种性质为*具有原子性的失败*（failure atomicity）：只要对象构造完毕，就不会再发生由值可变性引发的失败。

实现一个值不可变的 Java 类，我们需要做到以下事情。

- *把所有的字段都标记为* `final`。

  Java 要求被标记为 `final` 的字段，要么在声明时初始化，要么在构造器中初始化。不要在意 IDE 大惊小怪地提醒我们字段没有在声明位置上初始化，当我们在构造器里写好相关的初始化代码，IDE 就会明白过来。

- *把类标记为* `final`*，防止被子类覆盖。*

  如果类可以被覆盖，类中的方法也就有可能被改变行为，因此以防万一，我们干脆禁止子类化。Java 的 `String` 类就采取了这样的防范策略。

- *不要提供无参数的构造器。*

  一个值不可变的对象，它的一切状态都必须通过构造器来设定。假如我们没有需要设定的状态，那建立这么一个对象又有何必要呢？在无状态的类里面安排几个静态方法就足够了。所以说，值不可变的类根本不应该出现无参数的构造器。假如我们受到某些框架的限制，不得不提供无参数的构造器，这时可以考虑能否用一个私有的无参数构造器来满足框架的要求（私有的构造器仍然可以通过反射来访问）。

  JavaBeans 的标准规定要有默认构造器，我们摒除无参数构造器违反了这条规定。不过反正 JavaBeans 里有各种 `setXXX` 方法存在，本身就不可能是值不可变的。

- *提供至少一个构造器。*

  构造器是我们在对象里添置状态的最后机会！

- *除了构造器之外，不要提供任何制造变化的方法。*

  我们不但要避免沿袭 JavaBeans 风格的 `setXXX` 方法，还必须小心防范，不能返回任何值可变的对象引用。标记了 `final` 的对象引用并不等于它所指向的一切都不可改变。因此，我们需要预防性地复制所有通过 `getXXX` 方法返回的对象引用。

Groovy 用语法糖衣掩盖了实现值不可变性的繁琐细节，如例 7-4 所示。

例 7-4　值不可变的 `Client` 类

```
@Immutable
class Client {
    String name, city, state, zip
    String[] streets
}
```

添加 `@Immutable` 标注使得这个类自动获得以下特性。

- 它成为一个 `final` 类。
- 自动为属性生成相应的私有字段和 `get` 方法。
- 任何企图修改属性值的操作都会得到 `ReadOnlyPropertyException`。
- Groovy 会生成普通的和基于 map 的两种构造器。
- 集合类的实例会被加上适当的包装，数组（以及其他可被复制的对象）会被复制。
- 自动生成默认的 `equals()`、`hashcode()` 和 `toString()` 方法。

`@Immutable` 标注生动地阐释了本书屡屡强调的要旨：把实现细节托付给运行时。

对象－关系映射等编程工具往往以值可变的对象为前提，当它们面对值不可变对象的时候，要么效率低下，要么根本就不可行。因此，现有系统需经过多方面的重大改造，才可能全面地投向函数式的编程范式。

使用 Scala、Clojure 等函数式语言，以及它们的配套框架，比较容易搭建出深刻贯彻函数式内涵的系统。

也有一些架构立足于现有的基础设施，去追求理想的函数式境界，例如命令－查询职责隔离（Command-Query Responsibility Segregation，CQRS）架构。

CQRS

CQRS 的概念由 Greg Young（http://codebetter.com/gregyoung）提出，并在 Martin Fowler 撰文解说（http://dwz.cn/fowler-cqrs）之后扩大了影响。CQRS 架构从多个方面体现了函数式的观念。

传统应用程序架构把读数据和写数据交织在一起，典型的例子如关系型数据库的数据读写。开发者投入了无数的时间和精力在对象－关系映射上，却成果平平。图 7-1 画出了传统的应用程序架构。

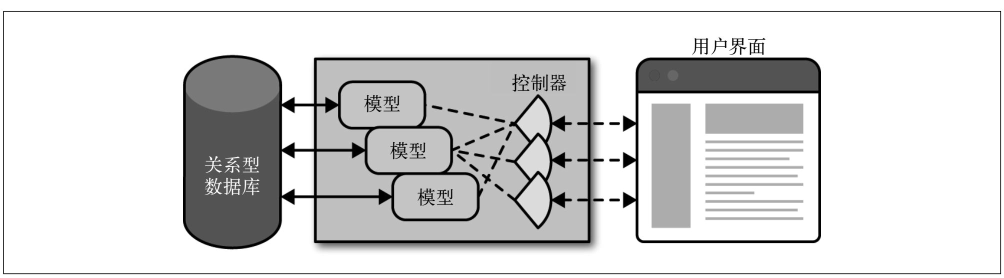


图 7-1：传统的应用程序架构

程序中的模型部分负责处理业务规则和验证等方面的工作。一般模型对象还要在其内部，或通过另外的逻辑层来协调持久化事项。开发者必须时时兼顾读、写两方面的潜在影响，

增加了复杂性。

CQRS 通过分离架构中负责读取的部分和负责命令的部分，部分地简化了程序的架构。如图 7-2 所示的 CQRS 场景，一部分模型（以及同样分化了功能的控制器）处理数据库的更新，另外一些模型负责数据的展示和报告。

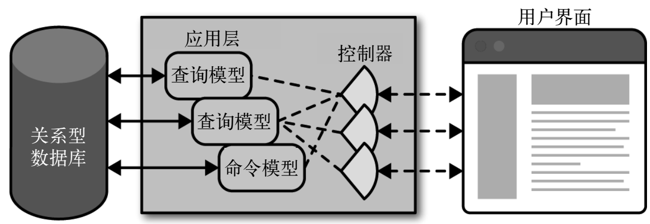


图 7-2：CQRS 架构

查询方面的逻辑一般较为简单，因为开发者可以在值不可变的前提下开展设计；更新则需要由专门的机制去处理。分离的模型暗示了它们的逻辑处理过程也是分离的，甚至连运行的机器都是分离的。例如我们可以安排一批服务器专门负责显示，而用户发出的修改和增加操作则被传递到另一个子网下的服务器。

架构永远是取舍的结果。CQRS 简化了一些方面，同时又复杂化了另一些方面。例如对于集中式的数据库来说，事务并不难处理。可是在 CQRS 架构下，我们很可能需要用最终一致性模型来取代事务性的模型。

最终一致性

分布式计算中的最终一致性（eventual consistency）模型，不对模型的变更操作施加硬性的时间限制，而只是保证，当更新发生后，模型最终会回复到一致的状态。

事务要求系统满足 ACID（即原子性 Atomic、一致性 Consistent、隔离性 Isolated、持久性 Durable 的缩写）性质，而最终一致性要求满足 BASE（即基本可用 Basically Available、软状态 Soft state、最终一致性 Eventual consistency 的缩写）性质。

放弃事务常常是应用程序在扩张规模时不得已的选择。CQRS 可以和 Event Sourcing 架构模式配合无间，该模式将应用程序所有的状态变化都汇集记录到一个事件流中。读取与变更分离之后，逻辑可得到简化。例如承担读取职责的部分可以全面实现值不可变的性质。

7.2.2 Web框架

讨论新语言和新范式的话题，没提到 Web 框架都不算完整。Web 领域与函数式编程简直是天作之合：我们可以把整个 Web 看作是一系列从请求到响应的变换。 每一种函数式语

言下都有非常丰富的，或热门或冷门的 Web 框架可供选择。这些 Web 框架大多具备以下共同特性。

- *路由框架*

  大多数现代的 Web 应用框架，包括函数式的 Web 框架在内，都从应用的主体功能中剥离路由的相关细节，将之交托给专门的路由功能库。路由信息通常放置在一个能够被路由库理解和遍历的标准数据结构里（例如存放在另一个 `Vector` 结构中的 `Vector` 结构）。

- *以函数作为路由的目标*

  很多函数式的 Web 应用用函数来充当路由的目的地。Web 请求很容易被理解成一个接受 `Request`，返回 `Response` 的函数；有一部分函数式 Web 框架提供了语法糖衣来简化这方面的基本操作。

- *领域专用语言*（DSL）

  Martin Fowler 将 DSL 定义为表达能力有限，专门针对一个狭窄问题域的计算机编程语言。其中内部 DSL 使用较为广泛。内部 DSL 是在其宿主语言之上构造出来的新“语言”，且利用宿主语言的语法糖衣来形成自身的风格。Ruby on Rails Web 框架（http://rubyonrails.org/）和 C# 语言的 LINQ 扩展（http://dwz.cn/ms-linq）都是内部 DSL 的好示范。

  大多数现代 Web 框架会在路由、HTML 内嵌元素、数据库交互等几个地方使用 DSL。DSL 在各类语言中都是常见的编程手段，但函数式语言尤其偏好描述性的代码风格，这一点恰好也常常是 DSL 的目的。

- *与构建工具紧密集成*

  函数式 Web 框架通常不和 IDE 拴在一起，反倒和命令行的构建工具紧密集成，用构建工具来执行从生成新项目骨架到运行测试的一切任务。IDE 和编辑器可以围绕现有的工具链来实现自动化，但构建工具和构建产物之间的耦合关系太过紧密，很难拆解之后重新安置。

函数式语言下的 Web 开发会有一些变得更容易的部分，也会有一些变得更困难的部分，这是通用语言的通病。总的来说，用不可变的值来编程，会降低测试的负担，因为需要通过测试来验证的状态变化减少了。一旦我们在架构中树立起（值不可变性等）惯例，各部分的配合将更加顺畅。

7.2.3 数据库

关系型数据库出于什么样的动机，要采取破坏性的方式来实现其更新操作呢？换言之，每次我们执行数据库更新操作的时候，在置入新值的同时，旧的值就被毁弃了。为什么数据库要设计成这个样子呢？是为了最大化地利用存储空间，以免数据无限制地增长下去。这个架构决策已经在数据库设计里扎根了几十年，可是现在世界变了。资源（尤其是虚拟化

的资源）变得十分廉价，以至于 Amazon 愿意用以分为单位的单价把资源租给我们！可是开发者却还在苦苦承受旧时代延续下来的架构约束。

Clojure 社群（http://clojure.org/）正在稳步地搭建一套为函数式架构提供支持的工具链，涵盖从浏览器一直到持久化层的广大范围。Datomic（http://www.datomic.com/）是 Clojure 社群进入商用 NoSQL 数据库领域的一次尝试。这个项目有意思的地方在于，它的架构设计每一处都浸染了函数式的概念，因此可以作为我们观察函数式概念应用极限的一个样本。

Datomic 是一种值不可变的数据库，进入到库里的每一笔事实都会被打上时间戳。在 Datomic 的记录看来，奥巴马成为总统，并不会抹消小布什曾经是总统的事实。由于 Datomic 存储值而非数据，它的空间利用效率并不低。当一个值被输入到系统之后，所有需要使用该值的地方都可以指向原始的实例（而该原始实例永远不会变更，因为它是值不可变的），这样就提高了空间的利用效率。如果未来需要向应用程序呈现更新的数据，我们只要指向另一个值就可以了。Datomic 在信息上增加了时间的概念，使得每一笔事实都总是维持在正确的上下文里。

Datomic 的设计产生了一些有意思的结果。

- *永久地记录所有的schema变更和数据变更*
  由于保留了一切事物的记录（包括 schema 的变动），数据库可以轻易地回退到旧的版本。而关系型数据库为了解决这个问题而发展出了一个完整的工具门类（数据库迁移工具）。

- *读取和写入分离*
  Datomic 的架构天然地分离了读取和写入操作，也就是说绝不会发生因为查询而推迟更新的情况。因此从架构上说，Datomic 拥有一个 CQRS 系统的内在。

- *事件驱动型架构中的值不可变性和时间戳*
  事件驱动的架构依靠一个事件流来反映应用程序的状态变化，而一个捕获所有信息并加上时间戳记的数据库，正好可以完美地扮演事件流的角色，数据库本身的特性即可满足回退和重放事件的需求。

Datomic 让我们看到，经过函数式编程深刻的范式转变之后，摆脱旧思维束缚的开发者有能力建造出像 Datomic 这样闪亮的工具和框架。

第8章

多语言与多范式

函数式编程是一种编程范式，它既是从特定角度去看待问题的思维框架，又是实现思维图景的配套工具。现代编程语言常常是多范式的，支持多种多样的编程范式，如面向对象、元编程、函数式、过程式，等等。

Groovy 是一种多范式的语言，它同时支持面向对象、元编程、函数式这几种大体上互相“正交”的风格。元编程可用于在语言及其核心库上添加额外的特性。把它和函数式编程结合在一起，可以获得更充分地实践函数式的代码风格，也可以用来增强第三方的函数式库，使之更好地配合 Groovy 语言本身的特性。下一节我们要展示如何通过 `ExpandoMetaClass` 来实施元编程，并使用这种方式将一个第三方函数式库（Functional Java）无缝地织入 Groovy 的语言核心。

> **正交**
>
> 正交（orthogonality）的概念被应用到很多学科里，其中包括数学和计算机科学。数学上把两个互相垂直的向量称作正交的，也就是说这两个量不相关。而在计算机科学里，两个组件如果互相没有任何影响（或副作用），就可以称作是正交的。例如在 Groovy 语言里，函数式编程和元编程是正交的，因为它们不会互相干扰：使用元编程并不妨碍我们使用函数式编程的语言构造，反之亦然。请不要将两种范式的正交关系理解成它们不能共存，它们只是不会互相干扰。

8.1 函数式与元编程的结合

我们又要把完美数分类的例子拿出来了。例子到目前为止的实现虽然很实用，但形态上仍然是由一个个函数构成的。假设完美数分类是我们业务上的一个关键要素，那么为了使用方便，我们也许希望它能够成为编程语言的一部分。Groovy 可以实现这个目标，通过它的 `ExpandoMetaClass` 机制，我们可以在已有的类，包括语言运行时提供的类上添加方法。例 8-1 依照第 2 章中完美数分类的函数式实现，在 `Integer` 类上添加了相关方法，使之具备了完美数分类的能力。

例 8-1　通过元编程使 `Integer` 具备完美数分类能力

```
Integer.metaClass.isPerfect = {->
  Classifier.isPerfect(delegate)
}

Integer.metaClass.isAbundant = {->
  Classifier.isAbundant(delegate)
}

Integer.metaClass.isDeficient = {->
  Classifier.isDeficient(delegate)
}
```

例 8-1 在 `Integer` 上添加了三个来自 `Classifier` 的方法。这样一来，所有的 Groovy 整数实例都会拥有这几个方法。Groovy 没有所谓的“基本数据类型”，就连常量也是用 `Integer` 类型来定义的。例中定义方法的几个代码块都使用了 `delegate` 关键字来获得方法调用者的对象实例，也就是待分类的整数值。

> **初始化经元编程添加的方法**
>
> Groovy 要求我们在调用经元编程添加的方法之前，必须先对其初始化。最安全的初始化位置，是在使用这些方法的类的静态初始化块里（因为可以确保在类的其他初始化过程之前执行），但是当需要使用这些方法的类比较多的时候，初始化的工作会成为增加复杂性的一个负担。因此频繁使用元编程的程序通常会安排一个专门的引导类，以确保在正确的时间完成初始化。

初始化经元编程添加的几个方法之后，我们就可以让数字自己来“回答”它是否属于某个完美数分类了：

```
assertTrue num.isDeficient()
assertTrue 6.isPerfect()
```

新的方法在变量和常量上都是有效的。我们很容易在 `Integer` 上再增加一个方法，让它直接返回一个枚举值来表示数字所属的分类。

在现有类上添加新方法的做法本身谈不上有多么“函数式”，就算添加进来的代码函数式色彩再浓厚也一样。但是，当我们拥有了无缝添加新方法的能力之后，就可以轻松地将一些第三方库，如 Functional Java（http://functionaljava.org/）吸收合并进来，获得其丰富的函数式特性。

8.2 利用元编程在数据类型之间建立映射

Groovy 本质上是一种 Java 方言，因此导入像 Functional Java 这样的第三方库不存在任何障碍。不过，假如我们能够借助一些元编程的手段，在库中的数据类型和 Groovy 语言的数据类型之间建立映射关系的话，将减少接缝部位的龃龉，双方也就结合得更加紧密。Groovy 拥有内建的闭包类型（`Closure` 类）。Functional Java 则没有那么幸运（因为要遵循 Java 5 语法），其作者不得不依赖泛型和一个通用的、含有 `f()` 方法的 `F` 类来达到目的。我们可以利用 Groovy 的 `ExpandoMetaClass` 机制，在 Groovy 的闭包和 Functional Java 的泛型方法之间建立映射，使双方不相匹配的数据类型吻合起来。

我们首先改造 Functional Java 的 `Stream` 类，它是对无限长度的列表的抽象。我们打算把 Functional Java 原设计在参数位置上传递的 `F` 实例，替换成 Groovy 的闭包。为了实现这种对接，我们重载了 `Stream` 类的几个方法来建立闭包和 `F` 类 `f()` 方法之间的映射，如例 8-2 所示。

例 8-2 通过元编程将 Functional Java 的类映射成集合

```
static {
  Stream.metaClass.filter = { c -> delegate.filter(c as fj.F) }
  // Stream.metaClass.filter = { Closure c -> delegate.filter(c as fj.F) }
  Stream.metaClass.getAt = { n -> delegate.index(n) }
  Stream.metaClass.getAt = { Range r -> r.collect { delegate.index(it) } }
}

@Test
void adding_methods_to_fj_classes() {
  def evens = Stream.range(0).filter { it % 2 == 0 }
  assertTrue(evens.take(5).asList() == [0, 2, 4, 6, 8])
  assertTrue((8..12).collect { evens[it] } == [16, 18, 20, 22, 24])
  assertTrue(evens[3..6] == [6, 8, 10, 12])
}
```

例 8-2 的第一行在 `Stream` 上添加了新的 `filter()` 方法，其参数是一个闭包（即定义代码块的参数 `c`）。（被注释掉的）第二行与第一行实质相同，仅仅增加了 `Closure` 类型声明；这种写法不会影响代码执行，但对于说明代码的意义有正面的作用。新方法在其定义中将 Groovy 闭包映射成 Functional Java 的 `fj.F` 类之后，以其为参数调用了 `Stream` 原有的 `filter()` 方法。Groovy 神奇的 `as` 运算符为我们完成了映射。

Groovy 的 `as` 运算符可以让闭包形态的“方法”和接口所要求的方法一一对应起来，强令

其满足接口的定义。请看例 8-3 的演示。

例 8-3　利用 Groovy 的 as 运算符将 map 结构强行映射成特定接口的实现

```
h = [hasNext: { h.i > 0 }, next: {h.i--}]
h.i = 10    // ❶
def pseudoIterator = h as Iterator    // ❷

while (pseudoIterator.hasNext())
  print pseudoIterator.next() + (pseudoIterator.hasNext() ? ", " : "\n")
// 10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
```

❶ 闭包可以引用 map 中的成员变量。

❷ as 运算符为接口制造了一个实现。

例 8-3 首先创建了一个散列结构，里面包含两个键值对。其中键的部分是一个字符串（Groovy 不要求用双引号把键名括起来，而是默认其为字符串），值的部分则是代码块。as 运算符对这个散列结构进行了映射，使之满足 Iterator 接口所要求的 hasNext() 和 next() 方法。经过映射，我们就可以把这个散列结构当成 Iterator 实例来使用了。假如我们需要映射的接口只有一个方法，或者我们希望将接口中的所有方法都映射到同一个闭包，那么可以舍弃散列结构，直接用 as 运算符把闭包映射到函数上。这方面的例子可见例 8-2 的第一行，该处将传入的闭包映射成了单方法的 F 类。例 8-2 还对两个 getAt() 方法做了映射（其中一个以数字为参数，另一个以 Range 实例为参数），因为筛选操作需要用到这两个方法。

改造后的 Stream 可以当作无限序列来使用，例 8-2 后半部分的测试演示了这样的用法。例中通过传给 filter 方法的闭包对整数序列进行了筛选，创建出一个从 0 开始的偶数序列。显然我们不能一次性取得无限序列中的所有元素，必须通过 take() 来取出想要的数目。例 8-2 的测试用例从不同角度展示了 Stream 的用法。

由Functional Java和Groovy共同构造的无限长序列

我们在第 4 章尝试过用 Groovy 来实现缓求值的无限长列表。我们能不能利用一下 Functional Java 的无限序列来代替原先的手工实现呢？

为了实现无限长的完美数序列，我们还需要改造 Stream 类的两个方法，让它们接受用 Groovy 闭包来做参数，如例 8-4 所示。

例 8-4　实现完美数序列需要改造的两个方法

```
  static {
    Stream.metaClass.asList = { delegate.toCollection().asList() }
//    Stream.metaClass.static.cons =
//      { head, Closure c -> delegate.cons(head, ['_1':c] as fj.P1)}
    Stream.metaClass.static.cons =
```

```
        { head, closure -> delegate.cons(head, closure as fj.P1) }
}
```

例 8-4 首先添加了一个转换方法 `asList()`，通过它可以方便地将 Functional Java 的 `Stream` 对象转换成 `List` 对象。另一个被改造的是 `Stream` 赖以构造新序列的 `cons()` 方法的其中一个重载版本。我们用来表示无限长列表的数据结构，一般有一个作为列表头部的首元素，还有一个作为列表尾部的闭包块，当它被调用时就会产生列表的下一个元素。为了实现 Groovy 版的无限长完美数序列，我们需要让 Functional Java 能够理解作为参数传给 `cons()` 方法的 Groovy 闭包。

`as` 运算符可以将单个闭包映射到含有多个方法的接口上，在这种情况下，不管调用接口中的哪个方法，都会执行相同的闭包。像这样简单的映射方式足以应付大多数的 Functional Java 类。只有一小部分方法要求传入 `fj.P1()` 而非 `fj.F`。假如下游的方法完全不会用到 `P1` 的其他函数，那么我们还是可以用简单的映射来勉强应付过去。只有在需要精确映射的时候，例如像例 8-4 被注释掉的代码行那样，需要将 `_1()` 方法映射到闭包上来创建一个散列，我们才使用更复杂的映射方式。虽然 `_1()` 方法看起来比较怪异，但它其实是 `fj.P1` 类的一个标准方法，用来返回第一个元素。

当我们完成对 `Stream` 类的元编程改造，添加了必要的方法之后，就可以用它和例 4-16 的 Groovy 版完美数分类程序一起，创建一个无限长的完美数序列，如例 8-5 所示。

例 8-5　由 Functional Java 和 Groovy 共同构造的无限长完美数序列

```
import static fj.data.Stream.cons
import static com.nealford.ft.allaboutlists.NumberClassification.
nextPerfectNumberAfter

def perfectNumbers(num) {
    cons(nextPerfectNumberAfter(num), { perfectNumbers(nextPerfectNumberAfter(num))})
}

@Test
void infinite_stream_of_perfect_nums_using_funtional_java() {
    assertEquals([6, 28, 496], perfectNumbers(1).take(3).asList())
}
```

为了简化代码，例中静态导入了 Functional Java 的 `cons()` 方法和例 4-16 的 `nextPerfect-NumberAfter()` 方法。`perfectNumbers()` 方法首先找到种子数之后的第一个完美数，用它作为首元素，和作为次元素的闭包一起，“cons”出一个无限长的完美数序列。闭包返回的还是一个无限序列，序列的头部是下一个完美数，尾部则是用来计算再下一个数的闭包。例中的测试首先生成一个从数字 `1` 开始的完美数序列，然后从中取出前三个数与样本列表进行对比。

开发者们一般只会考虑使用元编程来编写自己的代码，很少会想到用它来改造别人的代码。Groovy 不仅允许我们在 `Integer` 这种内建类上添加方法，还可以把元编程手段运用到

像 Functional Java 这样的第三方库上。元编程与函数式编程结合之后，可以在短小的代码中注入巨大的力量，形成外表不着痕迹的紧密联系。

我们完全可以直接在 Groovy 代码中调用 Functional Java 的类，但比起真正的闭包，Functional Java 的许多构件显得过于笨拙累赘。因此我们通过元编程的映射手段，令 Groovy 便利的数据结构可以直接用在 Functional Java 的方法上，使双方都发挥出各自最大的优势。当项目更多、更深地牵涉到多语言的时候，开发者会经常需要在不同语言的不同类型之间施行类似的映射，例如 Groovy 的闭包和 Scala 的闭包在字节码层面上完全是两种东西。Java 8 的标准化工作有望推动这类对接问题下放到运行时层面去解决，使本节演示的映射手法失去存在的意义，但在达到理想状态之前，还是要靠这些手段来帮助我们写出简洁有力的代码。

8.3 多范式语言的后顾之忧

多范式的语言赋予了开发者组合、搭配不同范式的余裕，这是多范式语言潜藏的力量。Java 8 以前的旧版 Java 束手束脚令很多开发者着恼，而像 Groovy 这样的语言则拥有包括元编程和函数式的语言构造在内，丰富得多的编程手段可供驱策。

多范式语言虽然强大，但也要求开发者更注重纪律，才能驾驭好大型的项目。由于语言支持各式各样的抽象和观念，由不同开发者群体制作出来的库也会呈现明显的差异。我们在第 6 章已经讨论过，不同范式的基本思路就不一样。举例来说，代码重用在面向对象的世界里多表现为对结构的重用，而函数式的世界则多以组合和高阶函数作为重用的手段。如果要为公司设计一套 `Customer` API，你会选择什么样的风格呢？很多从 Java 语言迁移到 Ruby 语言的开发者都遇到了这个难题，因为 Ruby 是一款兼容并包的多范式语言。

依靠工程纪律来保证所有的开发者都朝着同一个方向努力，是解决协调问题的一种途径。单元测试为开发者精确地理解经元编程实现的复杂扩展提供了方便。开发者可以运用“消费者驱动的契约”等技巧，（以测试的形式）在不同团队之间建立可执行的契约。

> **消费者驱动的契约**
>
> 消费者驱动的契约（consumer-driven contract，http://dwz.cn/fowler-cd-contract）是由一项集成工作的实施方与各组件供应方共同商定的一组测试。集成实施方“同意”将该组测试纳入为日常构建过程的一部分，并确保所有的测试始终为绿色通过状态。测试维护着作为相关各方共识的前设条件。如果有某一方需要打破前设条件，那么必须召集所有受影响的当事方，议定一组新的测试。在这样的安排下，消费者驱动的契约既为集成提供了一重可执行的安全保障，同时又仅在有演变必要的时候产生协调事务。

过程式风格与面向对象风格夹杂不清的现象，对大量的 C++ 项目造成了损害。所幸现代的

工程实践和过去防范多范式语言风险的经验能够给我们一点帮助。

有些语言选择将一种范式树立为主导范式来避免风格混乱，再从实际出发去支持其他的范式。例如 Clojure 稳稳地守住它作为 JVM 上的函数式 Lisp 方言的定位。它为开发者操作（和新建）底层 JVM 平台上的类和方法提供了方便，然而值不可变性、缓求值等函数式意味浓厚的范式才是它支持的重心。Clojure 的开发实践还离不开包括测试在内的工程纪律的规正，但只要按照习惯用法去做，就不会偏离 Clojure 语言的立意太远。

8.4 上下文型抽象与复合型抽象的对比

函数式思维不只体现在项目采用的语言上，它还影响到工具的设计。我在第 6 章说过，复合是函数式编程领域奉为圭臬的设计原则。现在我想把它推广到工具的领域，并对比两种在编程世界中流行的抽象形态，一种是复合型（composable）抽象，另一种是上下文型（contextual）抽象。基于插件的架构可以作为上下文型抽象的代表。开发者可以直接从插件 API 处继承得到，或者间接透过已有的方法来号令插件 API 提供林林总总的数据结构和其他有用的上下文。但是为了使用 API，开发者必须首先理解上下文提供的是什么，而“理解”有时候是十分昂贵的。我询问过开发者在使用编辑器或 IDE 的过程中，会不会经常对 IDE 本身的行为做一些较大的、至少不能靠调整设置来完成的改动。结果即使是天天离不开 IDE 的重度用户也很少做这样的事情，因为扩展 Eclipse 之类的工具必须跨过极高的知识门槛。不起眼的改动和完成改动所需的知识、精力之间的落差，让开发者永远地打消了磨砺工具的念头。上下文型的工具绝不是什么坏事，没有这样的架构设计，就不会有 Eclipse 和 IntelliJ。上下文型的工具为开发者事先准备了极丰富的基础设施。一旦掌握 Eclipse API 错综复杂的细节，开发者可以获得被 API 封装起来的巨大能量。可是这里面有一个大大的痛处：怎样封装？

20 世纪 90 年代后期，各种 4GL 语言（http://dwz.cn/wiki-4gl）一时风光无限，它们都是上下文式设计思路的范本。它们把上下文内化成了语言的一部分：dBASE、FoxPro、Clipper、Paradox、PowerBuilder、Microsoft Access 以及其他同类型平台，全部在语言和配套工具中直接内置了数据库的相关设施。4GL 语言最终由于 Dietzler 定律而衰落，这条定律是我在 *The Productive Programmer*（O’Reilly 出版社，http://dwz.cn/oreilly-tpp）书中根据同事 Terry Dietzler 维护 Access 项目的经验总结出来的。

> **Dietzler 的 Access 定律**
>
> 所有 Access 项目最后都会失败，原因是，在用户想要的功能里，有 80% 实现起来既迅速又简单，还有 10% 能实现但较困难，而最后的 10% 是办不到的，因为不可能足够深地突破内建抽象去访问底层。可是，用户总想 100% 地满足需求。

Dietzler 定律让 4GL 语言最终丢掉了市场。虽然它们可以轻松而迅速地完成简单的任务，却无法作出足够的调整来适应现实中的需求。于是我们又退回到了通用语言。

复合型的系统倾向于使用一些细粒度的部件来组成整体，且这些部件都已准备好以特定方式相连接。我们可以在 Unix shell 中找到复合型抽象的鲜明范例，在 shell 命令行里，各种互相独立、五花八门的行为可以串联在一起创造出新的事物。我曾在第 1 章引述了发生在 1992 年的一个著名故事（http://dwz.cn/more-shell-less-egg）来展示函数式抽象的威力。当时 Donald Knuth 被要求写一段程序来解决这样一个文本操作问题：*读入一个文本文件，确定前 n 个使用频率最高的单词，并按照词频高低顺序打印出这些单词及其词频*。结果他写了一段超过 10 页代码的 Pascal 程序，而且在过程中设计（并记录）了一种新的算法。在 Knuth 的答案后面，Doug McIlroy 写下了篇幅不超过一条微博的 shell 脚本，轻松、优雅、容易理解（如果能看懂 shell 命令的话）地解决了相同的问题：

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

我疑心就连 Unix shell 的设计者也常常会感到惊讶，他们作出来的简单但有强大组合能力的抽象，到了开发者们的手里，竟会出现那么多创造性的用法。

上下文型的系统提供了更多扶持性的“脚手架”设施，更完善的预设行为，以及“脚手架”上承载的上下文智能。可以说，上下文型系统更多地依靠周全的预设功能来缓解开发者初次使用时的不顺。在这类系统中，继承关系有时候会隐含地带入一些全局性的庞大数据结构，成为下游派生扩展的巨大包袱。复合型的系统较少隐性的行为，较容易上手，但倾向于提供细粒度的构造单元，需经过一定过程才能发挥真正的实力。设计得当的复合型系统应当在封装的模块内提供窄范围的、局部的上下文。

这两种抽象方式也被运用到工具和框架的设计上，尤其是一些解决问题的能力必须随着项目的复杂性增长而增长的工具，如构建工具。在经历过深刻的教训之后，我们现在知道，*复合型的构建工具（对时间、复杂度、用途）的适应性要好于上下文型的构建工具*。上下文型的工具如 Ant 和 Maven 都准备了用于扩展的插件 API，因此在原作者预见范围之内的扩展实现起来并不困难。可是，如果想要突破 API 的设计框架去做一些预料之外的扩展，那么难度就会直线上升，甚至完全不可能了——Dietzler 定律再次现出身影。当工具履行其功能的关键信息，如构建任务的执行次序，无法通过常规手段得到的时候，这种扩展上的窘迫表现得尤为明显。

这就是所有的项目最终都憎恶 Maven 的原因。Maven 是典型的上下文型工具，它武断死板、照本宣科，而这些正好是*项目开始的时候需要的禀性*。在项目还是一张白纸的阶段，

我们没理由拒绝硬塞过来的现成结构和预先建好的便利设施，例如可以轻松添加新行为的插件机制。但时过境迁，项目中可以照本宣科的地方会越来越少，反而像一个真正的项目那样千头万绪起来。一开始大家还不了解情况，对生命周期之类的事情还没有主意的时候，一套生硬的系统有它的好处。可是当项目发展到一定阶段，其复杂性逼迫着开发者必须依照自己的想法来解决问题，然而 Maven 式的工具并不在乎开发者自己的想法。

寄居在语言上的工具往往表现出更强的复合型特性。Ruby 世界的构建工具，Rake（http://rake.rubyforge.org/），是我最喜欢的专用构建语言，可以用在各种公私项目上（几乎不受项目本身技术选型的限制）。它是简洁与力量的奇妙结合。当我第一次从 Ant 迁移到 Rake 的时候，花了很多时间来翻查 Rake 的文档，想看看哪里列出了 Rake 支持的构建任务，就像 Ant 文档中常见的那种列出全部构建任务（和扩展）的长列表。徒劳的搜寻让我开始反感 Rake 文档的不完善，直到我想通了找不到这种文档的原因：我们可以在 Rake 的构建任务中做任何事情，因为它就是 Ruby 代码。Rake 所做的事情，除了增加一些方便操作文件列表的辅助工具之外，基本上专注于管理构建任务的依赖关系和日常维护，不去妨碍开发者的发挥。

有人可能认为我在贬低 Maven，其实不是的——我是想从原理上回答“什么时候应该用 Maven”的问题。没有一种工具能够完美地适用于一切场合，想要超出安全范围来使用工具的项目只会有吃不完的苦头。Maven 很适合新项目起步：它能保证一致性，丰富的预设功能又有极高的性价比。只不过起步做得好并不等于未来发展顺利。（实际上结果几乎总是相反的。）所以真正的诀窍是，开始先用 Maven，哪天它开始无理取闹了，我们就把它甩掉换新欢。因为只要跟 Maven 有了第一次的摩擦，就再也回不到初相识的玫瑰色时光了。

好在世界上至少还有 Gradle（http://www.gradle.org/）这个 Maven 过来人的救星，它能够理解项目中已有的 Maven 设置，而且它被实现为 Groovy 语言的一种内部 DSL，复合能力好于插件式设计的 Maven。

很多上下文型的系统在被重新设计成一种 DSL 之后，会发掘出更强的复合能力。Ruby on Rails 等类似框架所做的事情，其实和以前的 4GL 很相似，只是有一点关键的区别：它们都被实现成寄居在通用语言里的内部 DSL。当开发者在这些框架里遭遇到 Dietzler 定律的瓶颈的时候，他们可以绕过框架，直接在底层的通用语言上做文章。Rake 和 Gradle 不约而同地选择了 DSL 的形式，我也认为，构建脚本在不同项目中的差异性和独特性太高，并不适合使用上下文型的工具。

8.5 函数式金字塔

计算机语言一般可以根据类型的强与弱、静态与动态这两条坐标轴来确定其在分类图谱中的位置，如图 8-1 所示。

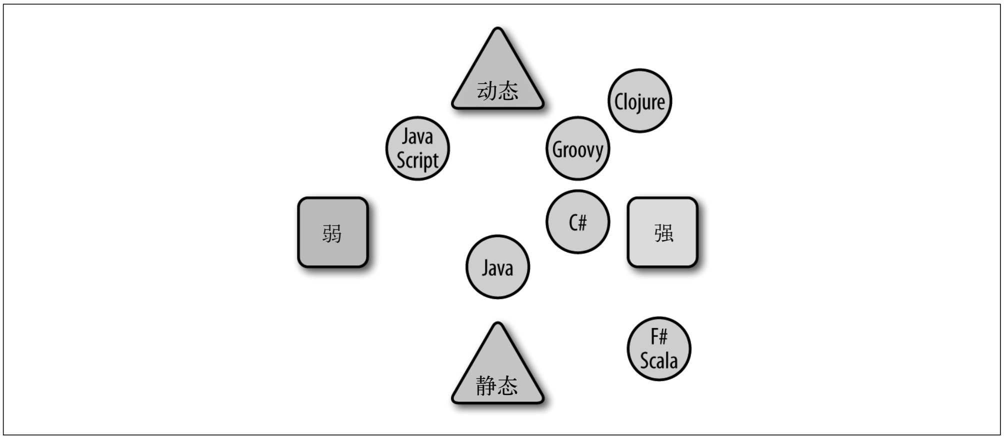


图 8-1：语言的分类

静态类型要求我们事先指定变量和函数的类型，而动态类型则允许推迟指定类型。强类型的变量“知道”自己的类型，允许反射和对实例作类型测试，且一直保有自身的类型信息。弱类型的语言相对不了解变量所指向的内容。例如，C 是静态的弱类型语言，全世界的 C 程序员都只能带着欣喜或者恐惧（或者兼而有之）的心情承认，C 语言中的变量本质上只是一组可以被任意解读的二进制位集。

Java 是强类型的静态语言，必须在声明变量的时候指定其类型，有时还要指定好几次。Scala、C# 和 F# 也都是强类型的静态语言，但它们的类型表述远没有 Java 那么繁冗，这是因为使用了类型推导。很多时候，语言能够辨别出正确的类型，于是我们可以省略掉多余的类型声明。

图 8-1 的分类图谱不是什么新东西；图中的两种区分角度和编程语言的研究历史一样长。可是现在我们又多了一个新的角度：*函数式编程*。

我们从本书反复的展示中知道，函数式编程语言的设计哲学不同于命令式语言。命令式语言希望让状态的修改变得更容易，并且围绕这个目的发展了众多的特性。而函数式语言希望尽可能减少可变的状态，因此更多地发展了通用性的计算设施。

不过，函数式范式并没有规定必须采用什么样的类型系统，如图 8-2 所示。

在增加了对值不可变性的依赖，甚至强制要求之后，区分语言的关键特征就不再是动态与静态之分了，而是落到了命令式与函数式这个维度上，这对我们构建软件的方式造成了深远的影响。

我在 2006 年所写的一篇文章不经意地让“多语言编程”（http://dwz.cn/ford-polyglot）这个词组重新流行了起来，并且赋予了新的含义：发挥现代语言运行时的优势，在同一运行时平台上，混合搭配不同的语言来创建应用。文章观点基于两个前提，一是 Java 和 .NET 平

台支持的语言加起来超过 200 种的事实，二是不存在一种“万能语言”能解决所有问题的假说。现代的托管运行时（managed runtime）赋予了我们在字节码层面上混合搭配不同语言的自由，我们可以主动根据任务的特点来选用最适当的语言。

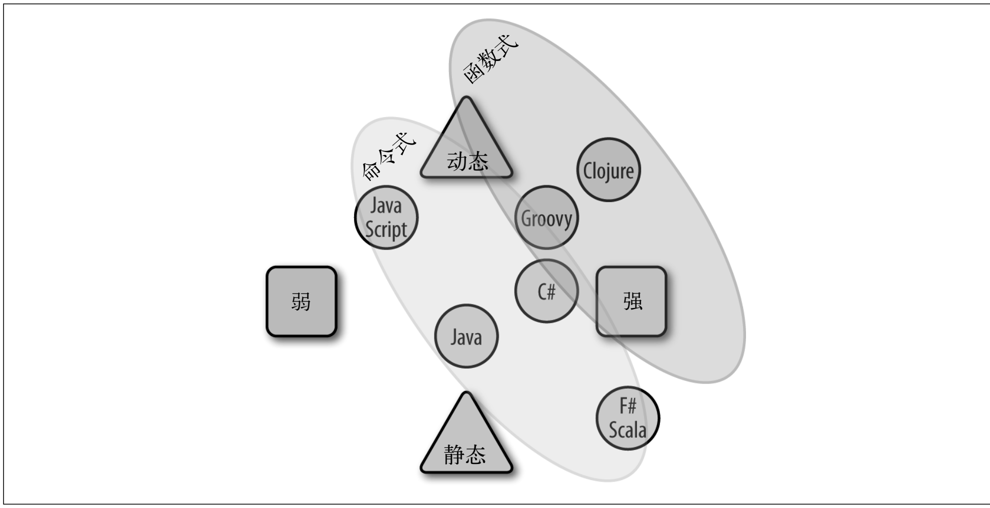


图 8-2：标示了范式的语言分类图谱

文章发表之后，我的同事 Ola Bini 也写了一篇文章作为回应，并在文中提出了他的“多语言金字塔”模型，该模型描述了人们在多语言世界中可能采用的一种应用架构，如图 8-3 所示。

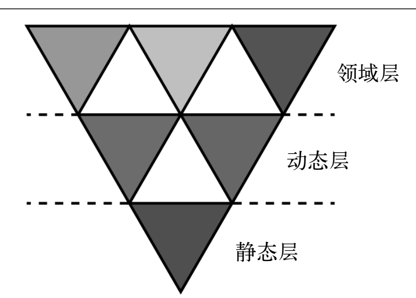


图 8-3：Ola Bini 的多语言金字塔

在 Ola Bini 的模型里，他建议使用静态语言来构建把可靠性排在第一位的最底层。在高一级应用层上，他的建议是使用更动态一些的语言，以利用其较为友好、简单的语法来完成用户界面等方面的构建。最后在模型最上方的是 DSL 层，开发者构造简洁的语言来封装重要的领域知识和工作流。DSL 一般采用动态语言来实现，因为它们的一些特性对实现工作较为有利。

这个金字塔比我最初的观点要深刻得多，不过我在重新审视现状之后，又对它做了一些修改。我的新观点认为，类型只是妨碍我们看清真相的干扰物，真正重要的其实是函数式与命令式这一对性质。因此我提出了一个新的多语言金字塔模型，如图 8-4 所示。

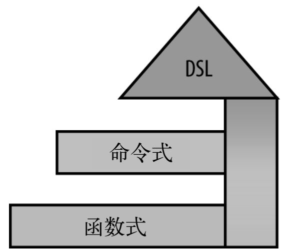


图 8-4：我的函数式金字塔

我认为，我们不应该从静态类型中求索程序抵御错误的能力，从根本上拥抱函数式的概念才是正确的方向。假如包括数据访问、集成等重要职责在内的所有核心 API，都能以值不变性为前提来设计的话，那么所有代码都会大幅度地简化。当然，在这种思路下，数据库和其他基础设施的构建方式也要随之发生变化，但我们知道最后结果一定会表现出由内而外的稳定性。

在函数式的内核之上，我们用命令式语言来编写系统中对开发效率要求较高的部分，例如工作流、业务规则、用户界面，等等。最上层和原来的模型一样，是 DSL 层，其作用也和原来一样。不过我觉得 DSL 会贯穿系统所有的层次，一直深入到最底层。论据是，有一些语言大大降低了 DSL 的实现门槛，如 Scala 语言（函数式、静态、强类型）和 Clojure 语言（函数式、动态、强类型），让我们轻松地借助 DSL 的简洁形式来表达重要的概念。

发生在应用架构模型上的变化是巨大的，而且意味深长。比起动态类型与静态类型的争拗，当前更有意义的讨论应该是对函数式风格与命令式风格的辨析，而范式转变的影响也比静态与动态之争更为深远。过去，我们在各种语言下承袭了命令式的设计。向着函数式风格的转变远不止学习新语法那么简单，但显著的成效是可以预见的。

作者简介

Neal Ford 在 ThoughtWorks 公司担任总监、软件架构师和文化基因传播人（Meme Wrangler）。ThoughtWorks 是一家专门针对软件开发和交付的全过程提供咨询服务的跨国 IT 企业。在加入 ThoughtWorks 之前，Neal 在美国知名的培训机构 The DSW Group, Ltd. 担任首席技术官。Neal 在佐治亚州立大学获得主修的计算机科学学位，以语言和编译器为研究方向，以及副修的数学学位，以统计分析为研究方向。他创作了各种应用程序、教学材料、杂志文章、演讲视频，还是多本书籍的作者，包括 *Developing with Delphi: Object-Oriented Techniques*（1996 年 Prentice-Hall 出版社）、*JBuilder 3 Unleashed*（1999 年 Sams 出版社，第一作者）、*Art of Java Web Development*（2003 年 Manning 出版社）、The *Productive Programmer*（2008 年 O'Reilly 出版社）、*Presentation Patterns*（2012 年 Pearson 出版社），且有众多文章被收入多部文集。Neal 的重点咨询业务是大规模企业应用的设计和构建。他还是一位国际知名的讲师，登上过全世界各种开发者会议的讲台。如果你对 Neal 产生了无法克制的好奇心，请访问他的网站 http://nealford.com/。Neal 欢迎读者的反馈，请通过 nford@thoughtworks.com 与他联系。

封面介绍

本书封面上的动物是大婴猴属的粗尾婴猴。这种灵长类动物分布在非洲南部和东部。基本树栖的婴猴喜欢选择热带和亚热带的森林作为栖息地，但有时候也可以在稀树草原上的林地里见到它们。

这种动物有着深褐色或者灰色的皮毛，耳朵和眼睛都很大，长长的尾巴可以帮助它们在枝杈间穿梭的时候保持平衡。它们还有很长的手指和脚趾，指尖的皮垫可以稳稳地抓住树枝。粗尾婴猴平均体长约 30 厘米（尾巴除外），平均体重约 0.9~1.4 公斤。

粗尾婴猴是夜行性动物。白天，它们在离地 5~12 米的地方歇息，树洞里密实的藤条小巢是隐蔽的藏身之所。粗尾婴猴通常独自生活，还会用胸部的气味腺和尿液来标记自己的领地（不过雄性的领地经常与雌性的领地重叠）。

晚上，婴猴们纷纷出来觅食。它们身手灵活，能小跳着从一棵树挂到另一棵树上，不过一般没有危险的时候，它们宁愿用走的。水果、种子、金合欢树胶、花、昆虫都是它们的食物。根据生物学家的观察，一个晚上，婴猴花在觅食上的时间只占 20%，将近一半的时间都在四处走动，并且经常是沿着同样的路线走动。

封面图案摘自 *Natural History*（Cassell 出版社）一书中的插图。

O'REILLY®

函数式编程思维

Java等现代编程语言中出现了越来越多的函数式特性，跟随这本书，去了解语法表象之下真正需要掌握的新思维。中高级开发者可以从知名软件架构师Neal Ford的演示中，体会到函数式编程思想是怎样通过改换视角，让我们站在了另一个抽象层次上，把编程问题看得更加清晰。

本书每一章都会给出各种函数式编程思维的示例，并用Java 8或其他具备函数式能力的JVM语言代码实现出来。改变你的思维是本书的愿望，至少读完本书的时候，你会对各种函数式概念有一个良好的把握。

具体说来，本书将——

- **解释为什么众多命令式语言都在增加函数式能力**
- **通过普通的编程问题来比较函数式和命令式的解答方案**
- **考察将例行杂务委托给运行时的各种方式**
- **学习用记忆和缓求值特性来取代手工编写的方案**
- **探讨在函数式语境下的设计模式和代码重用**
- **分别在Java 8、函数式架构和Web框架下检验函数式思维在真实案例中的表现**
- **分析生活在一个范式更丰富多彩的世界里的优缺点**

Neal Ford在跨国IT咨询公司ThoughtWorks担任总监、软件架构师和文化基因传播人。他精通各种编程语言，主要的咨询业务是大规模企业应用的设计、构建和工程实践。他还是一位国际知名的讲师，登上过全世界各种开发者会议的讲台。

“这是一本非常重要的书，而说到写这本书，没有人比Neal更合适了。”

——Venkat Subramaniam

Agile Developer公司总裁

SOFTWARE ENGINEERING

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议	计算机/软件开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-40041-3
9 787115 400413 >

ISBN 978-7-115-40041-3
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks